conference

proceedings

2011 USENIX Annual Technical Conference (USENIX ATC '11)

Portland, OR, USA
June 15–17, 2011

Proceedings of the 2011 USENIX Annual Technical Conference

Portland, OR, USA June 15–17, 2011

Sponsored by

USENIX

ISBN 978-931971-85-0

USENIX Association

Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC '11)

June 15–17, 2011
Portland, OR, USA

Conference Organizers

Program Co-Chairs

Jason Nieh, *Columbia University*
Carl Waldspurger

Program Committee

Keith Adams, *Facebook*
Saman Amarasinghe, *Massachusetts Institute of Technology*
Muli Ben-Yehuda, *Technion and IBM Research*
George Candea, *EPFL*
Leendert van Doorn, *AMD*
Dilma Da Silva, *IBM Research*
Dawson Engler, *Stanford University*
Úlfar Erlingsson, *Google*
Alexandra Fedorova, *Simon Fraser University*
Bryan Ford, *Yale University*
Dirk Grunwald, *University of Colorado at Boulder*
Ajay Gulati, *VMware*
Hermann Härtig, *Technische Universität Dresden*
Wilson Hsieh, *Google*
Angelos Keromytis, *Columbia University*
Eddie Kohler, *University of California, Los Angeles/ Meraki*
Monica Lam, *Stanford University*
Ed Nightingale, *Microsoft Research*
Kai Shen, *University of Rochester*
Alex C. Snoeren, *University of California, San Diego*
Ravi Soundararajan, *VMware*
Michael Swift, *University of Wisconsin—Madison*
Andy Tucker, *NetApp*
Volkmar Uhlig, *Teza Group*
Richard West, *Boston University*
Alec Wolman, *Microsoft Research*
Junfeng Yang, *Columbia University*
Erez Zadok, *Stony Brook University*

Poster Session Chair

Ajay Gulati, *VMware*

The USENIX Association Staff

External Reviewers

Gautam Altekar
Nadav Amit
Silviu Andrica
Jason Ansel
Radu Banabic
Orna Agmon Ben-Yehuda
Alexander Böttcher
Stefan Bucur
Mihai Budiu
Vitaly Chipounov
Christoffer Dall
Michael Dalton
Wim De Pauw
Stephan Diestelhorst
Björn Döbel
Ben Dodson
Benjamin Engel
Torsten Frenzel
Abel Gordon
Michael Gordon
Claude Hamann
Sudheendra Hangal
Nadav Har'El
Michael Hines
Michael Hohmuth
Horatiu Jula
Asim Kadav
Baris Kasikci
Ilia Kravets
Volodymyr Kuznetsov
John McCullough
Marek Olszewski
Una-May O'Reilly
Daniel Peek
Martin Pohlack
T.J. Purtell
Matthew Renzelmann
Michael Roitzsch
Kyung Ryu
Jiwon Seo
Marcio Silva
Julian Stecklina
Udo Steinberg
Jan Stoess
Dinesh Subhraveti
Hendrik Tews
Dan Tsafrir
Nicolas Viennot
Haris Volos
Marcus Völp
Carsten Weinhold
John Whaley
Jean Wolter
Weng-Fai Wong
Cristian Zamfir
Qin Zhao

2011 USENIX Annual Technical Conference
June 15–17, 2011
Portland, OR, USA

Wednesday, June 15

10:30–noon

1:00–2:30

3:00–4:30

Thursday, June 16

10:30–noon

1:00–2:30

3:00–4:30

Friday, June 17

8:30–9:30

10:00–noon

1:00–2:00

2:30–3:30

4:00–5:00

Message from the 2011 USENIX Annual Technical Conference Program Co-Chairs

Welcome to the 2011 USENIX Annual Technical Conference!

This year continues the USENIX Annual Tech tradition of papers and presentations reflecting some of the finest practical research in computer systems today. The program committee put together an excellent program of 27 full papers and 9 short papers, selected from 180 submissions. These papers reflect a broad range of work, including novel twists on core areas such as scheduling, storage, distributed systems, and virtualization, a renewed emphasis on important problems in debugging, diagnosis, security, and privacy, and new contributions in emerging areas such as cloud computing and personal devices.

It was our privilege to work with a great program committee of 28 members, from a range of industrial and academic research institutions. Program committee members were allowed to submit papers, but as program co-chairs, we did not submit any ourselves, to minimize complications and conflicts of interest. Every paper submitted to the conference received at least three reviews from program committee members, and every paper accepted to the conference received at least four reviews from program committee members. Overall, program committee members completed more than 700 reviews—roughly 25 reviews each. In addition, we benefited from the expertise of external reviewers who completed a number of additional reviews. The committee met at Columbia University on March 14, 2011, for an all-day discussion to decide the final program. Every program committee member except one attended the meeting. Nearly all of the accepted papers were then shepherded by a program committee member to deliver the final camera-ready version. Poster submissions were handled by Ajay Gulati, who served as the poster chair for the conference.

Many people deserve credit and thanks for their hard work in helping to make the conference successful. Without the many authors who submitted high-quality and thought-provoking papers, USENIX Annual Tech as a venue, event, and community of researchers would not exist. The program committee and our external reviewers devoted much time and diligence to reviewing and shepherding, in some cases on short notice. Eddie Kohler's HotCRP conference management software enabled the reviewing process and program committee meeting to run smoothly, and Tony Del Porto kept the USENIX servers up and running throughout the process. We are also grateful to Columbia University for hosting the program committee meeting, and to VMware, Yahoo!, and IBM for helping to sponsor the meeting. Finally, the USENIX staff—especially Ellie Young, Jane-Ellen Long, Casey Henderson, Camille Mulligan, Anne Dickison, Jessica Horst, and Andrew Gustafson—have worked tirelessly behind the scenes to make the conference a success.

We would like to thank our industry sponsors for their support in making the 2011 USENIX Annual Technical Conference possible and enjoyable. In particular, we thank our Silver Sponsors EMC, Facebook, and VMware, and our Bronze Sponsors Google, Microsoft Research, and NetApp, as well as our Media Sponsors and Industry Partners. Thanks also to NSF for providing student travel support for the conference.

We hope you enjoy the program and the conference!

Jason Nieh, ***Columbia University***
Carl Waldspurger

A Case for NUMA-aware Contention Management on Multicore Systems

Sergey Blagodurov
Simon Fraser University

Sergey Zhuravlev
Simon Fraser University

Mohammad Dashti
Simon Fraser University

Alexandra Fedorova
Simon Fraser University

Abstract

On multicore systems, contention for shared resources occurs when memory-intensive threads are co-scheduled on cores that share parts of the memory hierarchy, such as last-level caches and memory controllers. Previous work investigated how contention could be addressed via scheduling. A contention-aware scheduler separates competing threads onto separate memory hierarchy domains to eliminate resource sharing and, as a consequence, to mitigate contention. However, all previous work on contention-aware scheduling assumed that the underlying system is UMA (uniform memory access latencies, single memory controller). Modern multicore systems, however, are NUMA, which means that they feature non-uniform memory access latencies and multiple memory controllers.

We discovered that state-of-the-art contention management algorithms fail to be effective on NUMA systems and may even *hurt* performance relative to a default OS scheduler. In this paper we investigate the causes for this behavior and design the first contention-aware algorithm for NUMA systems.

1 Introduction

Contention for shared resources on multicore processors is a well-known problem. Consider a typical multicore system, schematically depicted in Figure 1, where cores share parts of the memory hierarchy, which we term *memory domains*, and compete for resources such as last-level caches (LLC), system request queues and memory controllers. Several studies investigated ways of reducing resource contention and one of the promising approaches that emerged recently is contention-aware scheduling [23, 10, 16]. A contention-aware scheduler identifies threads that compete for shared resources of a memory domain and places them into different domains. In doing so the scheduler can improve the worst-case performance of individual applications or threads by as much as 80% and the overall workload performance by as much as 12% [23].

Unfortunately studies of contention-aware algorithms focused primarily on UMA (Uniform Memory Access) systems, where there are multiple shared LLCs, but only a single memory node equipped with the single memory controller, and memory can be accessed with the same latency from any core. However, new multicore systems increasingly use the Non-Uniform Memory Access (NUMA) architecture, due to its decentralized and scalable nature. In modern NUMA systems, there are multiple memory nodes, one per memory domain (see Figure 1). Local nodes can be accessed in less time than remote ones, and each node has its own memory controller. When we ran the best known contention-aware schedulers on a NUMA system, we discovered that not only do they not manage contention effectively, but they sometimes even *hurt performance* when compared to a default contention-unaware scheduler (on our experimental setup we observed as much as 30% performance degradation caused by a NUMA-agnostic contention-aware algorithm relative to the default Linux scheduler). The focus of our study is to investigate (1) why contention-management schedulers that targeted UMA systems fail to work on NUMA systems and (2) devise an algorithm that would work effectively on NUMA systems.

Why existing contention-aware algorithms may hurt performance on NUMA systems: Existing state-of-the-art contention-aware algorithms work as follows on NUMA systems. They identify threads that are sharing a memory domain and hurting each other's performance and migrate one of the threads to a different domain. This may lead to a situation where a thread's memory is located in a different domain than that in which the thread is running. (E.g., consider a thread being migrated from core C1 to core C5 in Figure 1, with its memory being located in Memory Node #1). We refer to migrations that may place a thread into a domain remote from its mem-

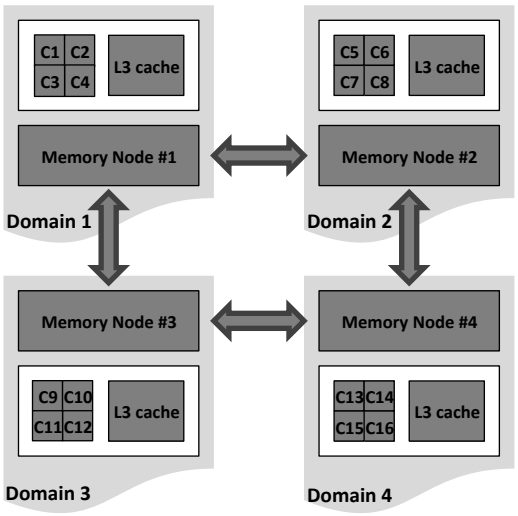


Figure 1: A schematic view of a system with four memory domains and four cores per domain. There are 16 cores in total, and a shared L3 cache per domain.

ory *NUMA-agnostic migrations*.

NUMA-agnostic migrations create several problems, an obvious one being that the thread now incurs a higher latency when accessing its memory. However, contrary to a commonly held belief that remote access latency – i.e., the higher latency incurred when accessing a remote domain relative to accessing a local one – would be the key concern in this scenario, we discovered that NUMA-agnostic migrations create other problems, which are far more serious than remote access latency. In particular, NUMA-agnostic migrations fail to eliminate contention for some of the key hardware resources on multicore systems and create contention for additional resources. That is why existing contention-aware algorithms that perform NUMA-agnostic migrations not only fail to be effective, but can substantially hurt performance on modern multicore systems.

Challenges in designing contention-aware algorithms for NUMA systems: To address this problem, a contention-aware algorithm on a NUMA system must migrate the memory of the thread to the same domain where it migrates the thread itself. However, the need to move memory along with the thread makes thread migrations costly. So the algorithm must minimize thread migrations, performing them only when they are likely to significantly increase performance, and when migrating memory it must carefully decide which pages are most profitable to migrate. Our work addresses these challenges.

The contributions of our work can be summarized as follows:

- We discover that contention-aware algorithms known to work well on UMA systems may actually *hurt* performance on NUMA systems.
- We identify NUMA-agnostic migration as the cause for this phenomenon and identify the reasons why performance degrades. We also show that remote access latency is not the key reason why NUMA-agnostic migration hurt performance.
- We design and implement *Distributed Intensity NUMA Online* (DINO), a new contention-aware algorithm for NUMA systems. DINO prevents superfluous thread migrations, but when it does perform migrations, it moves the memory of the threads along with the threads themselves. DINO performs up to 20% better than the default Linux scheduler and up to 50% better than Distributed Intensity, which is the best contention-aware scheduler known to us [23].
- We devise a page migration strategy that works online, uses Instruction-Based Sampling, and eliminates on average 75% of remote accesses.

Our algorithms were implemented at user-level, since modern operating systems typically export the interfaces for implementing the desired functionality. If needed, the algorithms can also be moved into the kernel itself.

The rest of this paper is organized as follows. Section 2 demonstrates why existing contention-aware algorithms fail to work on NUMA systems. Section 3 presents and evaluates DINO. Section 4 analyzes memory migration strategies. Section 5 provides the experimental results. Section 6 discusses related work, and Section 7 summarizes our findings.

2 Why existing algorithms do not work on NUMA systems

As we explained in the introduction, existing contention-aware algorithms perform NUMA-agnostic migration, and so a thread may end up running on a node remote from its memory. This creates additional problems besides introducing remote latency overhead. In particular, NUMA-agnostic migrations fail to eliminate *memory controller contention*, and create additional *interconnect contention*. The focus of this section is to experimentally demonstrate why this is the case.

To this end, in Section 2.1, we quantify how contention for various shared resources contributes to performance degradation that an application may experience as it shares the hardware with other applications. We show that memory controller contention and interconnect contention are the most important causes of performance degradation when an application is running remotely from its memory. Then, in Section 2.2 we use

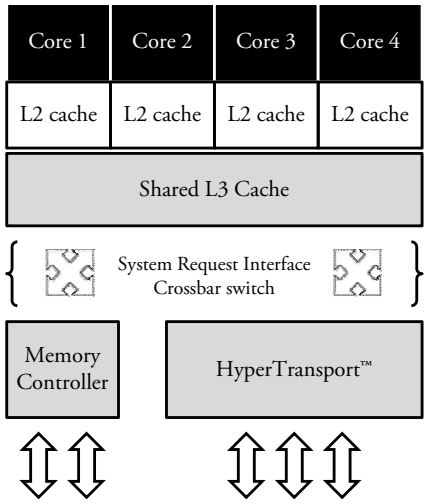


Figure 2: A schematic view of a system used in this study. A single domain is shown.

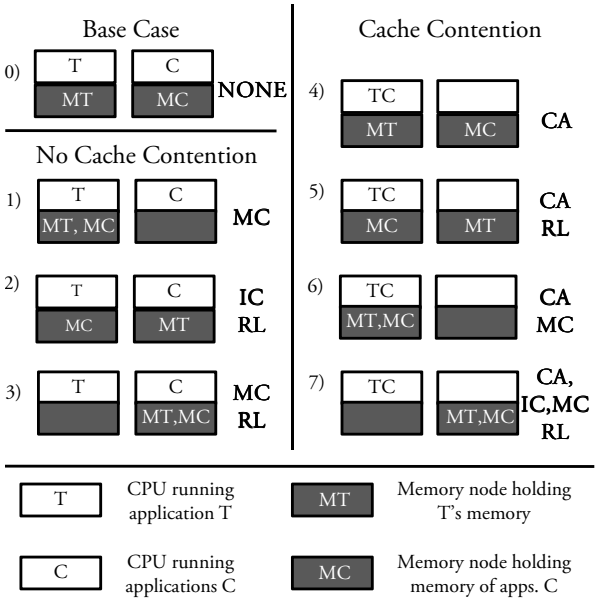


Figure 3: Placement of threads and memory in all experimental configurations.

this finding to explain why NUMA-agnostic migrations can be detrimental to performance.

2.1 Quantifying causes of contention

In this section we quantify the effects of performance degradation on multicore NUMA systems depending on how threads and their memory are placed in memory domains. For this part of the study, we use benchmarks from the SPEC CPU2006 benchmark suite. We perform experiments on a Dell PowerEdge server equipped with four AMD Barcelona processors running at 2.3GHz, and 64GB of RAM, 16GB per domain. The operating system is Linux 2.6.29.6. Figure 2 schematically represents the architecture of each processor in this system.

We identify four sources of performance degradation that can occur on modern NUMA systems, such as those shown in Figures 1 and 2:

- Contention for the shared last-level cache (CA). This also includes contention for the system request queue and the crossbar.
- Contention for the memory controller (MC). This also includes contention for the DRAM prefetching unit.
- Contention for the inter-domain interconnect (IC).
- Remote access latency, occurring when a thread's memory is placed in a remote node (RL).

To quantify the effects of performance degradation caused by these factors we use the methodology depicted in Figure 3. We run a target application, denoted as T with a set of three competing applications, denoted as C. The memory of the target application is denoted MT, and the memory of the competing applications is denoted MC. We vary (1) how the target application is placed with respect to its memory, (2) how it is placed with respect to the competing applications, and (3) how the memory of the target is placed with respect to the memory of the competing applications. Exploring performance in these various scenarios allows us to quantify the effects of NUMA-agnostic thread placement.

Figure 3 summarizes the relative placement of memory and applications that we used in our experiments. Next to each scenario we show factors affecting the performance of the target application: CA, IC, MC or RL. For example, in Scenario 0, an application runs contention-free with its memory on a local node, so no performance-degrading factors are present. We term this the *base* case and compare to it the performance in other cases. The scenarios where there is cache contention are shown on the right and the scenarios where there is no cache contention are shown on the left.

We used two types of target and competing applications, classified according to their memory intensity: *devil* and *turtle*. The terminology is borrowed from an earlier study on application classification [21]. Devils are memory intensive: they generate a large number of memory requests. We classify an application as a devil if it generates more than two misses per 1000 instructions (MPI). Otherwise, an application is deemed a turtle. We further divide devils into two subcategories: *regular devils* and *soft-devils*. Regular devils have a miss rate that exceeds 15 misses per 1000 instructions. Soft-devils

have an MPI between two and 15. Solo miss rates, obtained when an application runs on a machine alone, are used for classification.

We experimented with nine different target applications: three devils (*mcf*, *omnetpp* and *milc*), three soft-devils, (*gcc*, *bwaves* and *bzip*) and three turtles (*povray*, *calculix* and *h264*).

Figure 4 shows how an application's performance degrades in Scenarios 1-7 from Figure 3 relative to Scenario 0. Performance degradation, shown on the y-axis, is measured as the increase in completion time relative to Scenario 0. The x-axis shows the type of competing applications that were running concurrently to generate contention: devil, soft-devil, or turtle.

These results demonstrate a very important point exhibited in Scenario 3: when a thread runs alone on a memory node (i.e., there is no contention for cache), but its memory is remote and is in the same domain as the memory of another memory-intensive thread, performance degradation can be very severe, reaching 110% (see MILC, Scenario 3). One of the reasons is that the threads are still competing for the *memory controller* of the node that holds their memory. But this is exactly the scenario that can be created by a NUMA-agnostic migration, which migrates a thread to a different node without migrating its memory. This is the first piece of evidence showing why NUMA-agnostic migrations will cause problems.

We now present further evidence. Using the data in these experiments, we are able to estimate how much each of the four factors (CA, MC, IC, and RL) contributes to the overall performance degradation in Scenario 7 – the one where performance degradation is the worst. For that, we compare experiments that differ from each other precisely by one degradation factor involved. This allows us to single out the influence of this differentiating factor on the application performance. Figure 5 shows the breakdown for the devil and soft-devil applications. Turtles are not shown, because their performance degradation is negligible. The overall degradation for each application relative to the base case is shown at the top of the corresponding bar. The y-axis shows the fraction of the total performance degradation that each factor causes. Since contention causing factors on a real system overlap in complex and integrated ways, it is not possible to obtain a precise separation. These results are an approximation that is intended to direct attention to the true bottlenecks in the system.

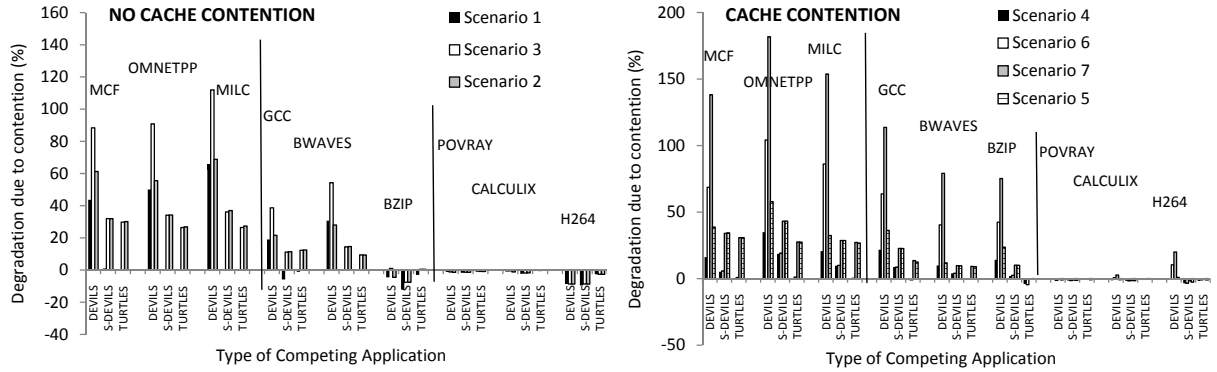
The results show that of all performance-degrading factors contention for cache constitutes only a very small part, contributing at most 20% to the overall degradation. And yet, NUMA-agnostic migrations eliminate only contention for the shared cache (CA), leaving the more important factors (MC, IC, RL) unaddressed! Since the memory is not migrated with the thread, several memory-intensive threads could still have their memory placed in the same memory node and so they would compete for the memory controller when accessing their memory. Furthermore, a migrated thread could be subject to the remote access latency, and because a thread would use the inter-node interconnect to access its memory, it would be subject to the interconnect contention. In summary, NUMA-agnostic migrations fail to eliminate or even exacerbate the most crucial performance-degrading factors: MC, IC, RL.

2.2 Why existing contention management algorithms hurt performance

Now that we are familiar with causes of performance degradation on NUMA systems, we are ready to explain why existing contention management algorithms fail to work on NUMA systems. Consider the following example. Suppose that two competing threads A and B run on cores C1 and C2 on a system shown in Figure 1. A contention-aware scheduler would detect that A and B compete and migrate one of the threads, for example thread B, to a core in a different memory domain, for example core C5. Now A and B are not competing for the last-level (L3) cache, and on UMA systems this would be sufficient to eliminate contention. But on a NUMA system shown in Figure 1, A and B are still competing for the memory controller at Memory Node #1 (MC in Figure 5), assuming that their memory is physically located in Node #1. So by simply migrating thread B to another memory domain, the scheduler does not eliminate one of the most significant sources of contention – contention for the memory controller.

Furthermore, the migration of thread B to a different memory domain creates two additional problems, which degrade thread B's performance. Assuming that thread B's memory is physically located in Memory Node #1 (all operating systems of which we are aware would allocate B's memory on Node #1 if B is running on a core attached to Node #1 and then leave the memory on Node #1 even after thread migration), B is now suffering from two additional sources of overhead: interconnect contention and remote latency (labeled IC and RL respectively in Figure 5). Although remote latency is not a crucially important factor, interconnect contention could hurt performance quite significantly.

To summarize, NUMA-agnostic migrations in the existing contention management algorithms cause the following problems, listed in the order of severity according to their effect on performance: (1) They fail to eliminate memory-controller contention; (2) They may create additional interconnect contention; (3) They introduce remote latency overhead.



(a) No cache contention

(b) Cache contention

Figure 4: Performance degradation due to contention, cases 1-7 from Figure 3 relative to running contention free (case 0).

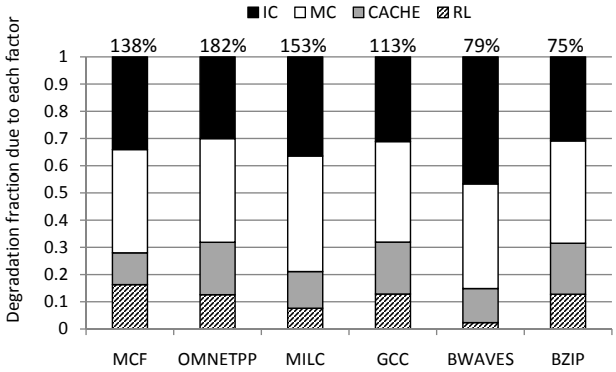


Figure 5: Contribution of each factor to the worst-case performance degradation.

3 A Contention-Aware Scheduling Algorithm for NUMA Systems

We design a new contention-aware scheduling algorithm for NUMA systems. We borrow the contention-modeling heuristic from the *Distributed Intensity* (DI) algorithm, because it was shown to perform within 3% percent of optimal on non-NUMA systems[1] [23]. Other contention aware algorithms use similar principles as DI [10, 16].

We begin by explaining how the original DI algorithm works (Section 3.1). For clarity we will refer to it from now on as *DI-Plain*. We proceed to show that simply extending DI-Plain to migrate memory – this version of the algorithm is called *DI-Migrate* – is not sufficient to achieve good performance on NUMA systems. We conclude with the description of our new *DI-NUMA Online*, or DINO, that in addition to migrating thread memory along with the thread eliminates superfluous migrations and unlike other algorithms improves performance on NUMA systems.

[1] Although some experiments with DI reported in [23] were performed on a NUMA machine, the experimental environment was configured so as to eliminate any effects of NUMA.

3.1 DI-Plain

DI-Plain works by predicting which threads will interfere if co-scheduled on the same memory domain and placing those threads on separate domains. Prediction is performed online, based on performance characteristics of threads measured via hardware counters. To predict interference, DI uses the *miss-rate heuristic* – a measure of last-level cache misses per thousand instructions, which includes the misses resulting from hardware prefetch requests. As we and other researchers showed in earlier work the miss-rate heuristic is a good approximation of contention: if two threads have a high LLC miss rate they are likely to compete for shared CPU resources and degrade each other's performance [23, 2, 10, 16].

Even though the miss rate does not capture the full complexity of thread interactions on modern multicore systems, it is an excellent predictor of contention for memory controllers and interconnects – key resource bottlenecks on these systems – because it reflects how intensely threads use these resources. Detailed study showing why the miss rate heuristic works well and how it compares to other modeling heuristics is reported in [23, 2].

DI-Plain continuously monitors the miss rates of running threads. Once in a while (every second in the original implementation), it sorts the threads according to their miss rates, and assigns them to memory domains so as to co-schedule low-miss-rate threads with high-miss-rate threads. It does so by first iterating over the sorted threads starting from the most memory-intensive (the one

with the highest miss rate) and placing each thread in a separate domain, iterating over domains consecutively. This way it separates memory-intensive threads. Then it iterates over the array from the other end, starting from the least memory-intensive thread, placing each on an unused core in consecutive domains. Then it iterates from the other end of the array again, and continues alternating iterations until all threads have been placed. This strategy results in balancing the memory intensity across domains. DI-Plain performs no memory migration when it migrates the threads.

Existing operating systems (Linux, Solaris) would not move the thread's memory to another node when a thread is moved to a new domain. Linux performs new memory allocations in the new domain, but will leave the memory allocated before migration in the old one. Solaris will act similarly[2]. So on either of these systems, if the thread after migration keeps accessing the memory that was allocated on another domain, it will cause negative performance effects described in Section 2.

3.2 DI-Migrate

Our first (and obvious) attempt to make DI-Plain NUMA-aware was to make it migrate the thread's memory along with the thread. We refer to this "intermediate" algorithm in our design exploration as *DI-Migrate*. The description of the memory migration algorithm is deferred until Section 4, but the general idea is that it detects which pages are actively accessed and migrates them to the new node along with a chunk of surrounding pages. For now we present a few experiments comparing DI-Plain with DI-Migrate. Our experiments will reveal that memory migration is insufficient to make DI-Plain work well on NUMA systems, and this will motivate the design of DINO.

Our experiments were performed on the same system as described in Section 2.1.

The benchmarks shown in this section are scientific applications from SPEC CPU2006 and SPEC MPI2007 suites with reference sets in both cases. (In a later section we also show results for the multithreaded Apache/MySQL workload.) We evaluated scientific applications for two reasons. First, they are CPU-intensive and often suffer from contention. Second, they were of interest for our partner Western Canadian Research Grid (WestGrid) – a network of compute clusters used by scientists at Canadian universities and in particular by physicists involved in ATLAS, an international particle physics experiment at the Large Hadron Collider at CERN. The WestGrid site at our university is interested in deploying contention management algorithms on their clusters. Prospect of adoption of contention management algorithms in a real setting also motivated their user-level implementation – not requiring a custom kernel makes the adoption less risky. Our algorithms are implemented on Linux as user-level daemons that measure threads' miss rates using `perfmon`, migrate threads using scheduling affinity system calls, and move memory using the `numa_migrate_pages` system call.

For SPEC CPU we show one workload for brevity; complete results are presented in Section 5. All benchmarks in the workload are launched simultaneously and if one benchmark terminates it is restarted until each benchmark completes three times. We use the result of the second execution for each benchmark, and perform the experiment ten times, reporting the average of these runs.

For SPEC MPI we show results for eleven different MPI jobs. In each experiment we run a single job, each comprised of 16 processes. We perform ten runs of each job and present the average completion times.

We compare performance under DI-Plain and DI-Migrate relative to the default Linux Completely Fair Scheduler, to which we refer as Default. Standard deviation across the runs is under 6% for the DI algorithms. Deviation under Default is necessarily high, because being unaware of resource contention it may force a low-contention thread placement in one run and a high-contention mapping in another. Detailed comparison of deviations under different schedulers is also presented in Section 5.

Figures 6 and 7 show the average completion time improvement for the SPEC CPU and SPEC MPI workloads respectively (higher numbers are better) under DI algorithms relative to Default. We draw two important conclusions. First of all, DI-Plain often *hurts* performance on NUMA systems, sometimes by as much as 36%. Second, while DI-Migrate eliminates performance loss and even improves it for SPEC CPU workloads, it fails to excel with SPEC MPI workloads, hurting performance by as much as 25% for GAPgeofem.

Our investigation revealed DI-Migrate migrated processes a lot more frequently in the SPEC MPI workload than in the SPEC CPU workload. While fewer than 50 migrations per process per hour were performed for SPEC CPU workloads, but as many as 400 (per process) were performed for SPEC MPI! DI-Migrate will migrate a thread to a different core any time its miss rate (and its position in the array sorted by miss rates) changes. For the dynamic SPEC MPI workload this happened rather frequently and led to frequent migrations.

[2]Solaris will perform new allocations in the new domain if a thread's home *lgroup* – a representation of a thread's home memory domain – is reassigned upon migration, but will not move the memory allocated prior to home lgroup reassignment. If the lgroup is unchanged, even new memory allocations will be performed in the old domain.

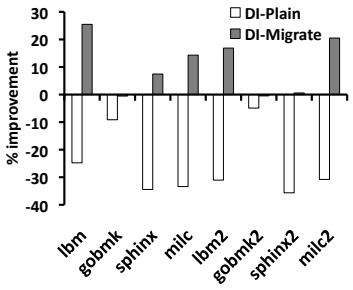


Figure 6: Improvement of completion time under DI-Plain and DI-Migrate relative to the Default for a SPEC CPU 2006 workload.

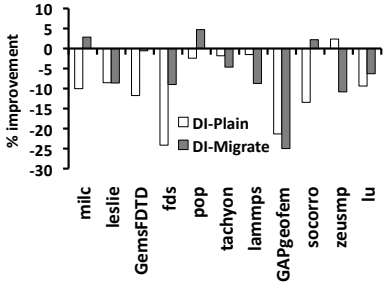


Figure 7: Improvement of completion time under DI-Plain and DI-Migrate relative to Default for eleven SPEC MPI 2007 jobs.

Unlike on UMA systems, thread migrations are not cheap on NUMA systems, because you also have to move the memory of the thread. No matter how efficient memory migrations are, they will never be completely free, so it is always worth reducing the number of migrations to the minimum, performing them only when they are likely to result in improved performance. Our analysis of DI-Migrate behaviour for the SPEC MPI workload revealed that oftentimes migrations resulted in a thread placement that was not better in terms of contention than the placement prior to migration. This invited opportunities for improvement, which we used in design of DINO.

3.3 DINO

3.3.1 Motivation

DINO's key novelty is in eliminating superfluous thread migrations – those that are not likely to reduce contention. Recall that DI-Plain (Section 3.1) triggers migrations when threads change their miss rates and their relative positions in the sorted array. Miss rates may change rather often, but we found that it is not necessary to respond to every change in order to reduce contention.

This insight comes from the observation that while the miss rate is an excellent heuristic for predicting relative contention at *coarse* granularity (and that is why it was shown to perform within 3% of the optimal oracular scheduler in DI) it does not perfectly predict how contention is affected by small changes in the miss rate.

Figure 8 illustrates this point. It shows on the x-axis SPEC CPU 2006 applications sorted in the decreasing order by their performance degradation when co-scheduled on the same domain with three instances of itself, relative to running solo. The bars show the miss rates and the line shows the degradations[3]. In general, with the exception of one outlier *mcf*, if one application has a much higher miss rate than another, it will have a much higher degradation. But if the difference in the miss rates is small, it is difficult to predict the relative difference in degradations.

What this means is that it is not necessary for the scheduler to migrate threads upon small changes in the miss rate, only upon the large ones.

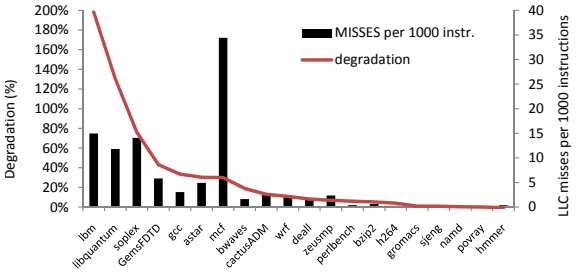


Figure 8: Performance degradation due to contention and miss rates for SPEC CPU2006 applications.

3.3.2 Thread classification in DINO and multi-threaded support

To build upon this insight, we design DINO to organize threads into broad *classes* according to their miss rates, and to perform migrations only when threads change their class, while trying to preserve thread-core affinities whenever possible. Classes are defined as follows (again, we borrow the animalistic classification from previous work):

Class 1: ***turtles*** – fewer than two LLC misses per 1000 instructions.
Class 2: ***devils*** – 2-100 LLC misses per 1000 instructions.
Class 3: ***super-devils*** – more than 100 LLC misses per 1000 instructions.

Threshold values for classes were chosen for our target architecture. Values for other architectures should be

[3]We omit several benchmarks whose counters failed to record during the experiment.

chosen by examining the relationship between the miss rates and degradations on that architecture.

Before we describe DINO in detail, we explain the special new features in DINO to deal with multithreaded applications.

First of all, DINO tries to co-schedule threads of the same application on the same memory domain, provided that this does not conflict with DINO's contention-aware assignment (described below). This assumes that performance improvement from co-operative data sharing when threads are co-scheduled on the same domain are much smaller than the negative effects of contention. This is true for many applications [22]. However, when this assumption does not hold, DINO can be extended to predict when co-scheduling threads on the same domain is more beneficial than separating them, using techniques described in [9] or [19].

When it is not possible to co-schedule all threads in an application on the same domain, and if threads actively share data, they will put pressure on memory controller and interconnects. While there is not much the scheduler can do in this situation (re-designing the application is the best alternative), it must at least avoid migrating the memory back and forth, so as not to make the performance worse. Therefore, DINO detects when the memory is being "ping-ponged" between nodes and discontinues memory migration in that case.

3.3.3 DINO algorithm description

We now explain how DINO works using an example.

In every rebalancing interval, set to one second in our implementation, DINO reads the miss rate of each thread from hardware counters. It then determines each thread's class based on its miss rate. To reduce the influence of sudden spikes, the thread only changes the class if it spent at least 7 out of the last 10 intervals with the missrate from the new class. Otherwise, the thread's class remains the same. We save this data as an array of tuples <new_class, new_processID, new_threadID>, sorted by memory-intensity of the class (e.g., super-devils, followed by devils and followed by turtles). Suppose we have a workload of eight threads containing two super-devils (D), three devils (d) and three turtles (t). Threads numbered <0, 3, 4, 5> are part of process 0. The remaining threads, numbered 1, 2, 6 and 7 each belong to a separate process, numbered 1, 2, 3 and 4 respectively [4]. Then the sorted tuple array will look like this:

```
new_class:     D D  d d d  t t t
new_processID: 0 4  0 2 3  0 0 1
new_threadID:  0 7  4 2 6  3 5 1
```

[4]DINO assigns a unique thread ID to each thread in the workload.

DINO then proceeds with the computation of *the placement layout* for the next interval. The placement layout defines how threads are placed on cores. It is computed by taking the most aggressive class instance (a super devil in our example) and placing it on a core in the first memory domain `dom0`, then the second aggressive (also a super devil) – on a core in the second domain and so on until we reach the last domain. Then we iterate from the opposite end of the array (starting with the least memory-intensive instance) and spread them across domains starting with `dom3`. We continue alternating between two ends of the array until all class instances have been placed on cores. In our example, for the NUMA machine with four memory domains and two cores per domain, the layout will be computed as follows:

```
domain:      dom0  dom1  dom2  dom3
new_core:    0 1   2 3   4 5   6 7
layout:      D t   D t   d t   d d
```

Although this example assumes that the number of threads equals the number of cores, the algorithm generalizes for scenarios when the number of threads is smaller or greater than the number of cores. In the latter case, each core will have T "slots" that can be filled with threads, where $T = num\_threads/num\_cores$, and instead of taking one class-instance from the array at a time, DINO will take T.

Now that we determined the layout for class-instances, we are yet to decide which thread will fill each core-class slot – any thread of the given class can potentially fill the slot corresponding to the class. In making this decision, we would like to match threads to class instances so as to *minimize the number of migrations*. And to achieve that, we refer to the matching solution for the old rebalancing interval, saved in the form of a tuple array: <old_domain, old_core, old_class, old_processID, old_threadID> for each thread.

Migrations are deemed superfluous if they change thread-core assignment, while not changing the placement of class-instances on cores. For example, if a thread that happens to be a devil (d) runs on a core that has been assigned the (d)-slot in the new assignment, it is not necessary to migrate this thread to another core with a (d)-slot. DI-Plain did not take this into consideration and thus performed a lot of superfluous migrations. To avoid them in DINO we first decide the thread assignment for any tuple that preserves core-class placement according to the new layout. So, if for a given thread `old_core = new_core` and `old_class = new_class`, then the corresponding tuple in the new solution for that thread will be <new_core, new_class, old_processID, old_threadID>.

For example, if the old solution were:

```
domain:          dom0  dom1  dom2  dom3
```

```
old_core:          0 1   2 3   4 5   6 7
old_class:         D t   d t   d t   d t
old_processID:     0 1   2 0   0 0   3 4
old_threadID:      0 1   2 3   4 5   6 7
```

then the initial shape of the new solution would be:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 1     0   0 0   3
new_threadID:      0 1     3   4 5   6
```

Then, the threads whose placement was not determined in the previous step – i.e., those whose old class is not the same as their current core's new class, as determined by the new placement, will fill the unused cores according to their new class:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 1   4 0   0 0   3 2
new_threadID:      0 1   7 3   4 5   6 2
```

Now that the thread placement is determined, DINO makes the final pass over the thread tuples to take care of multithreaded applications. For each thread A it checks if there is another thread B of the same multithreaded application (`new_processID(A) = new_processID(B)`) among the thread tuples not yet iterated so that B is not placed in the same memory domain with A. If there is one, we check the threads that are placed in the same memory domain with A. If there is a thread C in the same domain with A, such that `new_processID(A) != new_processID(C)` and `new_class(B) = new_class(C)` then we switch tuples B and C in the new solution. In our example this would result in the following assignment:

```
domain:            dom0  dom1  dom2  dom3
new_core:          0 1   2 3   4 5   6 7
new_class:         D t   D t   d t   d d
new_processID:     0 0   4 1   0 0   3 2
new_threadID:      0 3   7 1   4 5   6 2
```

DINO has complexity of $O(N)$ in the number of threads. Since the algorithm runs at most once a second, this has little overhead even for a large number of threads. We found that more frequent thread rebalancing did not yield better performance. Relatively infrequent changes of thread affinities mean that the algorithm is best suited for long-lived applications, such as the scientific applications we target in our study, data analytics (e.g., MapReduce), or servers. When there's more threads than cores coarse-grained rebalancing is performed by DINO, but fine-grained time sharing of cores between threads is performed by the kernel scheduler. If threads are I/O- or synchronization-intensive and have unequal sleep-awake periods, any resulting load imbalance must be corrected, e.g., as in [16].

3.3.4 DINO's Effect on Migration Frequency

We conclude this section by demonstrating how DINO is able to reduce migration frequency relative to DI-Migrate. Table 1 shows the average number of memory migrations per hour of execution under DI-Migrate and DINO for different applications from the workloads evaluated in Section 3.2. The results for MPI jobs are given for one of its processes and not for the whole job. Due to space limitations, we show the numbers for selected applications that are representative of the overall trend. The numbers show that DINO significantly reduces the number of migrations. As will be shown in Section 5, this results in up to 30% performance improvements for jobs in the MPI workload.

4 Memory migration

The straightforward solution to implement memory migration is to migrate the entire resident set of the thread when the thread is moved to another domain. This does not work for the following reasons. First of all, for multithreaded applications, even those where data sharing is rare, it is difficult to determine how the resident set is partitioned among the threads. Second, even if the application is single-threaded, if its resident set is large it will not fit into a single memory domain, so it is not possible to migrate it in its entirety. Finally, we experimentally found that even in cases where it is possible to migrate the entire resident set of a process, this can hurt performance of applications with large memory footprints. So in this section we describe how we designed and implemented a memory migration strategy that determines which of the thread's pages are most profitable to migrate when the thread is moved to a new core.

4.1 Designing the migration strategy

In order to rapidly evaluate various memory migration strategies, we designed a simulator based on a widely used binary instrumentation tool for x86 binaries called Pin [15]. Using Pin, we collected memory access traces of all SPEC CPU2006 benchmarks and then used a cache simulator on top of Pin to determine which of those accesses would be LLC misses, and so require an access to memory.

To evaluate memory migration strategies we used a metric called *Saved Remote Accesses* (SRA). SRA is the percent of the remote memory accesses that were eliminated using a particular memory migration strategy (after the thread was migrated) relative to not migrating the memory at all. For example, if we detect every remote access and migrate the corresponding page to the

Table 1: Average number of memory migrations per hour of execution under DI-Migrate and DINO for applications evaluated in Section 3.2.

	SPEC CPU2006					SPEC MPI2007				
	soplex	*milc*	*lbm*	*gamess*	*namd*	*leslie*	*lamps*	*GAPgeofem*	*socorro*	*lu*
DI-Migrate	36	22	11	47	41	381	135	237	340	256
DINO	8	6	5	7	6	2	1	3	2	1

thread's new memory node, we are eliminating all remote accesses, so the SRA would be 100%.

Each strategy that we evaluated detects when a thread is about to perform an access to a remote domain, and migrates one or more memory pages from the thread's virtual address space associated with the requested address. We tried the following strategies: ***sequential-forward*** where K pages including and following the one corresponding to the requested address are migrated; ***sequential-forward-backward*** where $K/2$ pages sequentially preceding and $K/2$ pages sequentially following the requested address are migrated; ***random*** where randomly chosen K pages are migrated; ***pattern-based*** where we detect a thread's memory-access pattern by monitoring its previous accesses, similarly to how hardware pre-fetchers do this, and migrate K pages that match the pattern. We found that sequential-forward-backward was the most effective migration policy in terms of SRA.

Another challenge in designing a memory migration strategy is minimizing the overhead of detecting which of the remote memory addresses are actually being accessed. Ideally, we want to be able to detect every remote access and migrate the associated pages. However, on modern hardware this would require unmapping address translations on a remote domain and handling a page fault every time a remote access occurs. This results in frequent interrupts and is therefore expensive.

After analyzing our options we decided to use hardware counter sampling available on modern x86 systems: PEBS (Precise Event-Based Sampling) on Intel processors and IBS (Instruction-Based Sampling) on AMD processors. These mechanisms tag a sample of instruction with various pieces of information; load and store instructions are annotated with the memory address.

While hardware-based event sampling has low overhead, it also provides relatively low sampling accuracy – on our system it samples less than one percent of instructions. So we also analysed how SRA is affected depending on the sampling accuracy as well as the number of pages that are being migrated. The lower the accuracy, the higher the value of K (pages to be migrated) needs to be to achieve a high SRA. For the hardware sampling accuracy that was acceptable in terms of CPU overhead (less than 1% per core), we found that migrating 4096 pages enables us to achieve the SRA as high as 74.9%. We also confirmed experimentally that this was a good value for K (results shown later).

4.2 Implementation of the memory migration algorithm

Our memory migration algorithm is implemented for AMD systems, and so we use IBS, which we access via Linux performance-monitoring tool `perfmon` [5].

Migration in DINO is performed in a user-level daemon running separately from the scheduling daemon. The daemon wakes up every ten milliseconds, sets up IBS to perform sampling, reads the next sample and migrates the page containing the memory address in the sample (if the sampled instruction was a load or a store) along with K pages in the application address space that sequentially precede and follow the accessed page. Page migration is effected using the *numa_move_pages* system call.

5 Evaluation

5.1 Workloads

In this section we evaluate DINO implemented using the migration strategy described in the previous section. We evaluate three workload types: SPEC CPU2006 applications, SPEC MPI2007 applications, and LAMP – Linux/Apache/MySQL/PHP.

We used two experimental systems for evaluation. One was described in Section 2.1. Another one is a Dell PowerEdge server equipped with two AMD Barcelona processors running at 2GHz, and 8GB of RAM, 4GB per domain. The operating system is Linux 2.6.29.6. The experimental design for SPEC CPU and MPI workloads was described in Section 3.2. The LAMP workload is described below.

The LAMP acronym is used to describe the application environment consisting of Linux, Apache, MySQL and PHP. The main data processing in LAMP is done by the Apache HTTP server and the MySQL database engine. The server management daemons *apache2* and *mysqld* are responsible for arranging access to the web-

site scripts and database files and performing the actual work of data storage and retrieval. We use Apache 2.2.14 with PHP version 5.2.12 and MySQL 5.0.84. Both *apache2* and *mysqld* are multithreaded applications that spawn one new distinct thread for each new client connection. This client thread within a daemon is then responsible for executing the client's request.

In our experiment, clients continuously retrieve from the Apache server various statistics about website activity. Our database is populated with the data gathered by the web statistics system for five real commercial websites. This data includes the information about website's audience activity (what pages on what website were accessed, in what order, etc.) as well as the information about visitors themselves (client OS, user agent information, browser settings, session id retrieved from the cookies, etc.). The total number of records in the database is more than 3 million. We have four Apache daemons, each responsible for handling a different type of request. There are also four MySQL daemons that perform maintenance of the website database.

We further demonstrate the effect that the choice of K (the number of pages that are moved on every migration) has on performance of DINO. Then we compare DINO to DI-Plain, DI-Migrate and Default.

5.2 Effect of K

Two of our workloads, SPEC CPU and LAMP demonstrate the key insights, and so we focus on those workloads. We show how performance changes as we vary the value of K. We compare to the scenario where DINO migrates the thread's entire resident set upon migrating the thread itself. The per-process resident sets of the two chosen workloads could actually fit in a single memory node on our system (it had 4GB per node), so whole-resident-set migration was possible. For SPEC CPU applications, resident sets vary from under a megabyte to 1.6GB for *mcf*. In general, they are in hundreds of megabytes for memory-intensive applications and much smaller for others. In LAMP, MySQL's resident set was about 400MB and Apache's was 120MB.

We show average completion time improvement (for Apache/MySQL this is average completion time per request), worst-case execution time improvement, and deviation improvement. Completion time improvement is the average over ten runs. To compute the worst-case execution time we run each workload ten times and record the longest completion time. Improvement in deviation is the percent reduction in standard deviation of the average completion time.

Figure 9 shows the results for the SPEC CPU workloads. Performance is hurt when we migrate a small number of pages, but becomes comparable to whole-resident-set migration when K reaches 4096. Whole-resident set migration actually works quite well for this workload, because migrations are performed infrequently and the resident set is small.

However upon experimenting with the LAMP workload we found that whole-resident set migration was detrimental to performance, most likely because the resident sets were much larger and also because this is a multithreaded workload where threads share data. Figure 10 shows performance and deviation improvement when $K = 4096$ relative to whole-resident-set migration. Performance is substantially improved when $K = 4096$. We experimented with smaller values of K, but found no substantial differences on performance.

We conclude that migrating very large chunks of memory is acceptable for processes with small resident sets, but not advisable for multithreaded applications and/or applications with large resident sets. DINO migrates threads infrequently, so a relatively large value of K results in good performance.

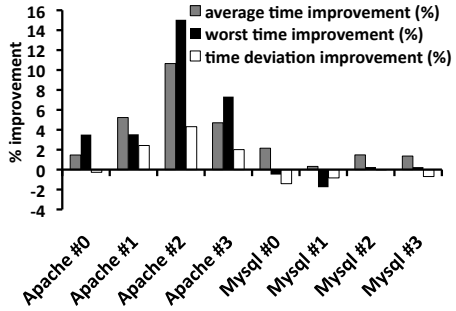
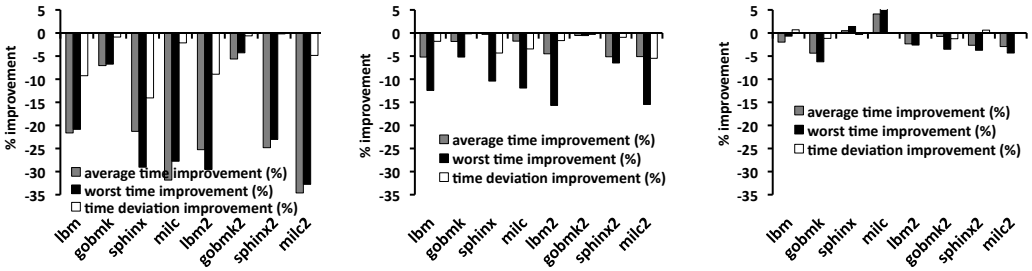


Figure 10: Performance improvement with DINO for $K = 4096$ relative to whole-resident-set migration for LAMP.

5.3 DINO vs. other algorithms

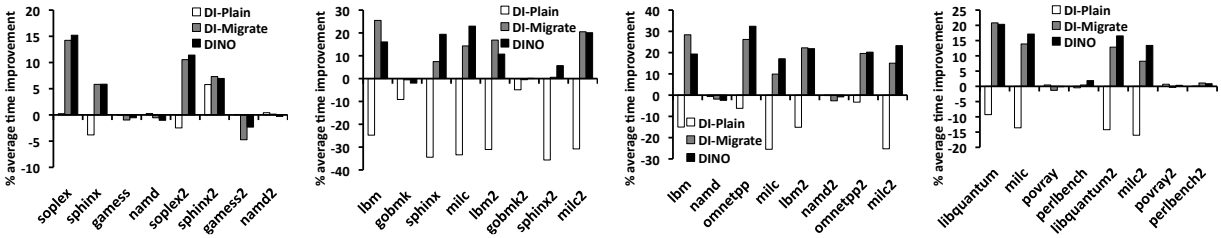
We compare performance under DINO, DI-Plain and DI-Migrate relative to Default, and similarly to the previous section, report completion time improvement, worst-case execution time improvement and deviation improvement.

Figures 11-13 show the results for the three workload types, SPEC CPU, SPEC MPI and LAMP respectively. For SPEC CPU, DI-Plain hurts completion time for many applications, but both DI-Migrate and DINO improve, with DINO performing slightly better than DI-Migrate for most applications. Worst-case improvement numbers show a similar trend, although DI-Plain does not perform as poorly here. Improvements in the worst-case execution time indicate that a scheduler is able to avoid pathological thread assignments that create especially high contention, and produce more stable performance. Deviation of running times is improved by all

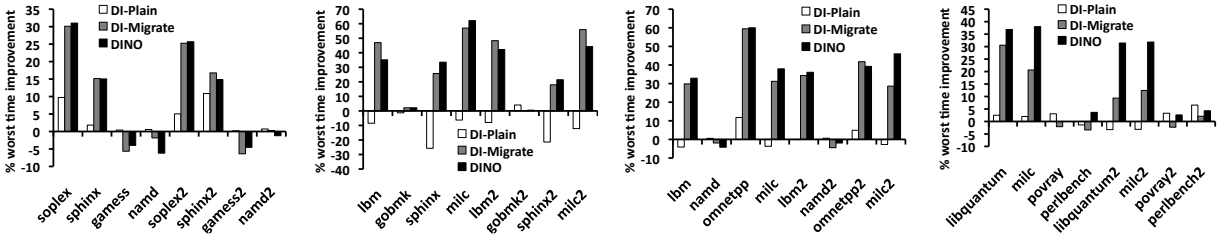


(a) 1 page(4KB) (b) 256 pages (1MB) (c) 4096 pages (16MB)

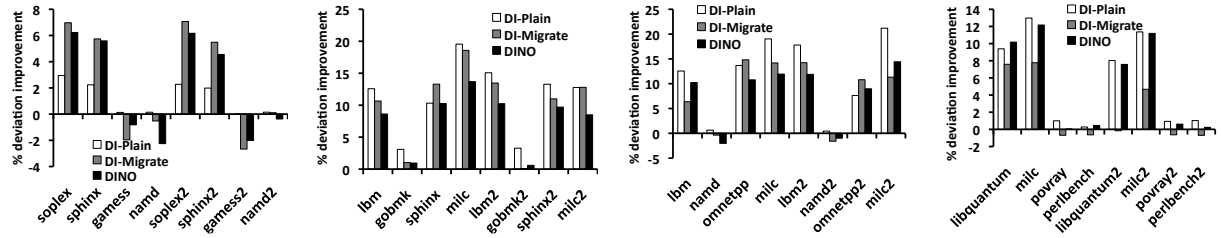
Figure 9: Performance improvement with DINO as K is varied relative to whole-resident-set migration for SPEC CPU.



(a) Average execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.



(b) Worst-case execution time improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.



(c) Deviation improvement of DINO (IBS), DI-Migrate (IBS) and DI-Plain over Default Linux scheduler.

Figure 11: DINO, DI-Migrate and DI-Plain relative to Default for SPEC CPU 2006 workloads.

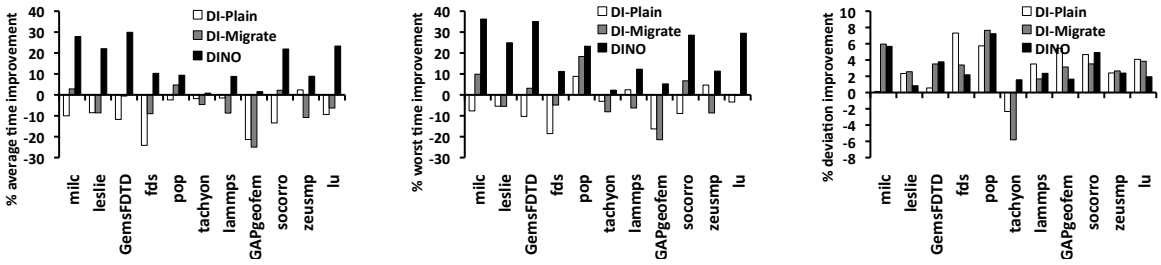
three schedulers relative to Default.

As to SPEC MPI workloads (Figure 12) only DINO is able to improve completion times across the board, by as much as 30% for some jobs. DI-Plain and DI-Migrate, on the other hand, can hurt performance by as much as 20%. Worst-case execution time also consistently improves under DINO, while sometimes degrading under DI-Plain and DI-Migrate.

LAMP is a tough workload for DINO or any scheduler that optimizes memory placement, because the workload is multithreaded and no matter how you place threads they still share data, putting pressure on interconnects. Nevertheless, DINO still manages to improve completion time and worst-case execution time in some cases, to a larger extent than the other two algorithms.

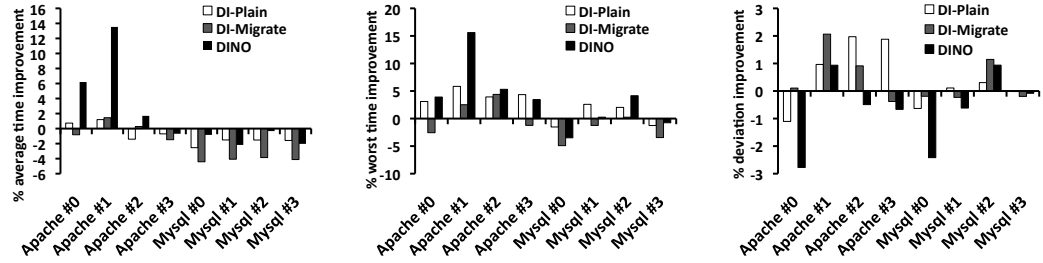
5.4 Discussion

Our evaluation demonstrates that DINO is significantly better at managing contention on NUMA systems than the DI algorithm designed without NUMA awareness or DI that was simply extended with memory migration.



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 12: DINO, DI-Migrate and DI-Plain relative to Default for SPEC MPI 2007.



(a) Completion time improvement (b) Worst-case time improvement (c) Deviation improvement

Figure 13: DINO, DI-Migrate and DI-Plain relative to Default for LAMP.

Multiprocess workloads representative of scientific Grid clusters show excellent performance under DINO. Improvements for the challenging multithreaded workloads are less significant as expected, and wherever degradation occurs for some threads it is outweighed by performance improvements for other threads.

6 Related Work

Research on NUMA-related optimizations to systems is rich and dates back many years. Many research efforts addressed efficient co-location of the computation and related memory on the same node [14, 3, 12, 19, 1, 4]. More ambitious proposals aimed to holistically redesign the operating system to dovetail with NUMA architectures [7, 17, 6, 20, 11]. None of the previous efforts, however, addressed shared resource contention in the context of NUMA systems and the associated challenges.

Li et al. in [14] introduced AMPS, an operating system scheduler for asymmetric multicore systems that supports NUMA architectures. AMPS implemented a NUMA-aware migration policy that can allow or deny thread migration requested by the scheduler. The authors used the *resident set size* of a thread in deciding whether or not the OS schedule is allowed to migrate thread to a different domain. If the migration overhead were expected to be high the migration would be disallowed. Our scheduler, instead of prohibiting migrations, detects which pages are being actively accessed and moves them as well as surrounding pages to the new domain.

LaRowe et al. [12] presented a dynamic multiple-copy policy placement and migration policy for NUMA systems. The policy periodically reevaluates its memory placement decisions and allows multiple physical copies of a single virtual page. It supports both migration and replication with the choice between the two operations based on reference history. A directory-based invalidation scheme is used to ensure the coherence of replicated pages. The policy applies a freeze/defrost strategy: to determine when to defrost a frozen page and trigger reevaluation of its placement is based on both time and reference history of the page. The authors evaluate various fine-grained page migration and/or replication strategies, however, since their test machine only has one processor per NUMA node, they do not address contention. The strategies developed in this work could have been very useful for our contention aware scheduler if the inexpensive mechanisms that the authors used for detecting page accesses were available to us. Detailed page reference history is difficult to obtain without hardware support; obtaining it in software may cause overhead for some workloads.

Goglin et al. [8] developed an effective implementation of the *move_pages* system call in Linux, which allows the dynamic migration of large memory areas to be significantly faster than in previous versions of the OS. This work is integrated in Linux kernel 2.6.29 [8], which we use for our experiments. The *Next-touch* pol-

icy, also introduced in the paper to facilitate thread-data affinity, works as follows: the application marks pages that it will likely access in the future as *Migrate-on-next-touch* using a new parameter to the `madvise()` system call. The Linux kernel then ensures that the next access to these pages causes a special page fault resulting in the pages being migrated to their threads. The work provides developers with an opportunity to improve memory proximity for their programs. Our work, on the other hand, improves memory proximity by using hardware counters data available on every modern machine. No involvement from the developer is needed.

Linux kernel since 2.6.12 supports the *cpusets* mechanism and its ability to migrate the memory of the applications confined to the cpuset along with their threads to the new nodes if the parameters of a cpuset change. Schermerhorn et al. further extended the cpuset functionality by adding an automatic page migration mechanism to it [18]: if enabled, it migrates the memory of a thread within the cpuset nodes whenever the thread migrates to a core adjacent to a different node. Two options for the memory migration are possible. The first is a lazy migration, when the kernel attempts to unmap any anonymous pages in the process's page table. When the process subsequently touches any of these unmapped pages, the swap fault handler will use the "migrate-on-fault" mechanism to migrate the misplaced pages to the correct node. Lazy migration may be disabled, in which case, automigration will use direct, synchronous migration to pull all anonymous pages mapped by the process to new node. The efficiency of lazy automigration is comparable to our memory migration solution based on IBS (we performed experiments to verify). However, automigration requires kernel modification (it is implemented as a collection of kernel patches), while our solution is implemented on user level. Cpuset mechanism needs explicit configuration from the system administrator and it does not perform contention management.

In [19] the authors group threads of the same application that are likely to share data onto neighbouring cores to minimize the costs of data sharing between them. They rely on several features of Performance Monitoring Unit unique to IBM Open-Power 720 PCs: the ability to monitor CPU stall breakdown charged to different microprocessor components and using the data sampling to track the sharing pattern between threads. The DINO algorithm introduced in our work complements [19] as it is designed to mitigate contention between applications. DINO provides sharing support by attempting to group threads of the same application and their memory on the same NUMA node, but as long as co-scheduling multiple threads of the same application does not contradict with a contention-aware schedule. In order to develop a more precise metric that assesses the effects of performance degradation versus the benefits from co-scheduling, we would need stronger hardware support, such as that available on IBM Open-Power 720 PCs or on the newest Nehalem systems (as demonstrated by the member of our team [9]).

The VMware ESX hypervisor supports NUMA load balancing and automatic page migration for its virtual machines (VMs) in commercial systems [1]. ESX Server 2 assigns each virtual machine a home node on whose processors a VM is allowed to run and its newly-allocated memory comes from the home node as well. Periodically, a special rebalancer module selects a VM and changes its home node to the least-loaded node. In our work we do not consider load balancing. Instead, we make thread migration decisions based on shared resource contention. To eliminate possible remote access penalties associated with accessing the memory on the old node, ESX Server 2 performs page migration from the virtual machine's original node to its new home node. ESX selects migration candidates based on finding hot remotely-accessed memory from page faults. The DINO scheduler, on the other hand, identifies hot pages using Instruction-Based Sampling. No modification to the OS is required.

The SGI Origin 2000 system [4] implemented the following hardware-supported [13] mechanism for co-location of computation and memory. When the difference between remote and local accesses for a given memory page is greater than a tunable threshold, an interrupt is generated to inform the operating system that the physical memory page is suffering an excessive number of remote references and hence has to be migrated. Our solution to page migration is different in that it detects "hot" remotely accessed pages via Instruction-Based Sampling, and performs migration in the context of a contention-aware scheduler.

In a series of papers [7] [17] [6] [20] the authors describe a novel operating system Tornado specifically designed for NUMA machines. The goal of this new OS is to provide data locality and application independence for OS objects thus minimizing penalties due to remote memory access in a NUMA system. The K42 [11] project, which is based on Tornado, is an open-source research operating system kernel that incorporates such innovative design principles like structuring the system using modular, object-oriented code (originally demonstrated in Tornado), designing the system to scale to very large shared-memory multiprocessors, avoiding centralized code paths and global locks and data structures and many more. K42 keeps physical memory close to where it is accessed. It uses large pages to reduce hardware and software costs of virtual memory. K42 project has resulted in many important contributions to Linux, on which our work relies. As a result, we were able to avoid

deleterious effects of remote memory accesses without requiring changes to the applications or the operating system. We believe that our NUMA contention-aware scheduling approach that was demonstrated to work effectively in Linux can also be easily implemented in K42 with its inherent user-level implementation of kernel functionality and native performance monitoring infrastructure.

7 Conclusions

We discovered that contention-aware algorithms designed for UMA systems may hurt performance on systems that are NUMA. We found that contention for memory controllers and interconnects occurring when thread runs remotely from its memory are the key causes. To address this problem we presented DINO: a new contention management algorithm for NUMA systems. While designing DINO we found that simply migrating a thread's memory when the thread is moved to a new node is not a sufficient solution; it is also important to eliminate superfluous migrations: those that add to migration cost without providing the benefit. The goals for our future work are (1) devising metric for predicting a trade-off between performance degradation and benefits from thread sharing and (2) investigate the impact of using small versus large memory pages during migration.

References

[1] VMware ESX Server 2 NUMA Support. White paper. *[Online] Available: http://www.vmware.com/pdf/esx2_NUMA.pdf*.

[2] BLAGODUROV, S., ZHURAVLEV, S., AND FEDOROVA, A. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst. 28* (December 2010), 8:1–8:45.

[3] BRECHT, T. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *USENIX SEDMS* (1993).

[4] CORBALAN, J., MARTORELL, X., AND LABARTA, J. Evaluation of the Memory Page Migration Influence in the System Performance: the Case of the SGI O2000. In *Proceedings of Supercomputing* (2003), pp. 121–129.

[5] ERANIAN, S. What can performance counters do for memory subsystem analysis? In *Proceedings of MSPC* (2008).

[6] GAMSA, B., KRIEGER, O., AND STUMM, M. Optimizing IPC Performance for Shared-Memory Multiprocessors. In *Proceedings of ICPP* (1994).

[7] GAMSA, B., KRIEGER, O., AND STUMM, M. Tornado: Maximizing Locality and Concurrency in a Shared Memory Multiprocessor Operating System. In *Proceedings of OSDI* (1999).

[8] GOGLIN, B., AND FURMENTO, N. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In *Proceedings of IPDPS* (2009).

[9] KAMALI, A. Sharing Aware Scheduling on Multicore Systems. Master's thesis, Simon Fraser University, 2010.

[10] KNAUERHASE, R., BRETT, P., HOHLT, B., LI, T., AND HAHN, S. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro 28*, 3 (2008), pp. 54–66.

[11] KRIEGER, O., AUSLANDER, M., ROSENBURG, B., WISNIEWSKI, R. W., XENIDIS, J., DA SILVA, D., OSTROWSKI, M., APPAVOO, J., BUTRICO, M., MERGEN, M., WATERLAND, A., AND UHLIG, V. K42: Building a Complete Operating System. In *Proceedings of EuroSys* (2006).

[12] LAROWE, R. P., JR., ELLIS, C. S., AND HOLLIDAY, M. A. Evaluation of NUMA Memory Management Through Modeling and Measurements. *IEEE Transactions on Parallel and Distributed Systems 3* (1991), 686–701.

[13] LAUDON, J., AND LENOSKI, D. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of ISCA* (1997).

[14] LI, T., BAUMBERGER, D., KOUFATY, D. A., AND HAHN, S. Efficient Operating System Scheduling for Performance-Asymmetric Multi-core Architectures. In *Proceedings of Supercomputing* (2007).

[15] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of PLDI* (2005).

[16] MERKEL, A., STOESS, J., AND BELLOSA, F. Resource-Conscious Scheduling for Energy Efficiency on Multicore Processors. In *Proceedings of EuroSys* (2010).

[17] PARSONS, E., GAMSA, B., KRIEGER, O., AND STUMM, M. (De-)Clustering Objects for Multiprocessor System Software. In *Proceedings of IWOOS* (1995).

[18] SCHERMERHORN, L. T. Automatic Page Migration for Linux.

[19] TAM, D., AZIMI, R., AND STUMM, M. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *Proceedings of EuroSys* (2007).

[20] UNRAU, R. C., KRIEGER, O., GAMSA, B., AND STUMM, M. Hierarchical Clustering: a Structure for Scalable Multiprocessor Operating System Design. *J. Supercomput. 9*, 1-2 (1995), 105–134.

[21] XIE, Y., AND LOH, G. Dynamic Classification of Program Memory Behaviors in CMPs. In *Proceedings of CMP-MSI* (2008).

[22] ZHANG, E. Z., JIANG, Y., AND SHEN, X. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *Proceedings of PPOPP* (2010).

[23] ZHURAVLEV, S., BLAGODUROV, S., AND FEDOROVA, A. Addressing Contention on Multicore Processors via Scheduling. In *Proceedings of ASPLOS* (2010).

TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments

Shinpei Kato[†] [‡], Karthik Lakshmanan[†], and Ragunathan (Raj) Rajkumar[†], Yutaka Ishikawa[‡]
† *Department of Electrical and Computer Engineering, Carnegie Mellon University*
‡ *Department of Computer Science, The University of Tokyo*

Abstract

The Graphics Processing Unit (GPU) is now commonly used for graphics and data-parallel computing. As more and more applications tend to accelerate on the GPU in multi-tasking environments where multiple tasks access the GPU concurrently, operating systems must provide prioritization and isolation capabilities in GPU resource management, particularly in real-time setups.

We present TimeGraph, a real-time GPU scheduler at the device-driver level for protecting important GPU workloads from performance interference. TimeGraph adopts a new event-driven model that synchronizes the GPU with the CPU to monitor GPU commands issued from the user space and control GPU resource usage in a responsive manner. TimeGraph supports two priority-based scheduling policies in order to address the trade-off between response times and throughput introduced by the asynchronous and non-preemptive nature of GPU processing. Resource reservation mechanisms are also employed to account and enforce GPU resource usage, which prevent misbehaving tasks from exhausting GPU resources. Prediction of GPU command execution costs is further provided to enhance isolation.

Our experiments using OpenGL graphics benchmarks demonstrate that TimeGraph maintains the frame-rates of primary GPU tasks at the desired level even in the face of extreme GPU workloads, whereas these tasks become nearly unresponsive without TimeGraph support. Our findings also include that the performance overhead imposed on TimeGraph can be limited to 4-10%, and its event-driven scheduler improves throughput by about 30 times over the existing tick-driven scheduler.

1 Introduction

The Graphics Processing Unit (GPU) is the burgeoning platform to support high-performance graphics and data-parallel computing, as its peak performance is exceeding 1000 GFLOPS, which is nearly equivalent of 10 times that of traditional microprocessors. User-end windowing systems, for instance, use GPUs to present a more *lively* interface that improves the user experience significantly through 3-D windows, high-quality graphics, and smooth transition. Especially recent trends on 3-D browser and desktop applications, such as SpaceTime, Web3D, 3D-Desktop, Compiz Fusion, BumpTop, Cooliris, and Windows Aero, are all intriguing possibilities for future user interfaces. GPUs are also leveraged in various domains of general-purpose GPU (GPGPU) processing to facilitate data-parallel compute-intensive applications.

Real-time multi-tasking support is a key requirement for such emerging GPU applications. For example, users could launch multiple GPU applications concurrently in their desktop computers, including games, video players, web browsers, and live messengers, sharing the same GPU. In such a case, quality-aware soft real-time applications like games and video players should be prioritized over live messengers and any other applications accessing the GPU in the background. Other examples include GPGPU-based cloud computing services, such as Amazon EC2, where virtual machines sharing GPUs must be prioritized and isolated from each other. More in general, important applications must be well-isolated from others for quality and security issues on GPUs, as on-line and user-space programs can create *any* arbitrary set of GPU commands, and access the GPU *directly* through generic I/O system calls, meaning that malicious and buggy programs can easily cause the GPU to be overloaded. Thus, GPU resource management consolidating prioritization and isolation capabilities plays a vital role in real-time multi-tasking environments.

GPU resource management is usually supported at the operating-system level, while GPU program code itself including GPU commands is generated through libraries, compilers, and runtime frameworks. Particularly, it is a device driver that transfers GPU commands from the CPU to the GPU, regardless of whether they produce graphics or GPGPU workloads. Hence, the development of a robust GPU device driver is of significant impact for many GPU applications. Unfortunately, existing GPU device drivers [1, 5, 7, 19, 25] are not tailored to support real-time multi-tasking environments, but accelerate *one* particular high-performance application in the system or provide *fairness* among applications.

We have conducted a preliminary evaluation to see the performance of existing GPU drivers, (i) the NVIDIA proprietary driver [19] and (ii) the Nouveau open-source

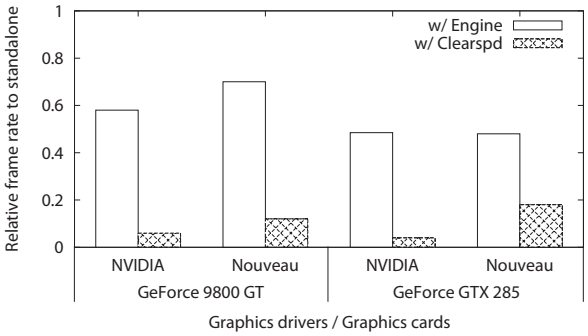


Figure 1: Decrease in performance of the OpenArena application competing with different GPU applications.

driver [7], in multi-tasking environments, using two different NVIDIA graphics cards, (i) GeForce 9800 GT and (ii) GeForce GTX 285, where the Linux 2.6.35 kernel is used as the underlying operating system. It should be noted that this NVIDIA driver evaluated on Linux is also expected to closely match the performance of the Windows driver (WDDM [25]), as they share about 90% of code [23]. Figure 1 shows the relative decrease in performance (frame-rate) of an OpenGL game (*OpenArena*) competing with two GPU-accelerated programs (*Engine* and *Clearspd* [15]) respectively. The *Engine* program represents a regularly-behaved GPU workload, while the *Clearspd* program produces a GPU command *bomb* causing the GPU to be overloaded, which represents a malicious or buggy program. To achieve the best possible performance, this preliminary evaluation assigns the highest CPU (`nice`) priority to the *OpenArena* application as an important application. As observed in Figure 1, the performance of the important *OpenArena* application drops significantly due to the existence of competing GPU applications. It highlights the fact that GPU resource management in the current state of the art is woefully inadequate, lacking prioritization and isolation capabilities for multiple GPU applications.

Contributions: We propose, design, and implement *TimeGraph*, a GPU scheduler to provide prioritization and isolation capabilities for GPU applications in *soft* real-time multi-tasking environments. We address a core challenge for GPU resource management posed due to the asynchronous and non-preemptive nature of GPU processing. Specifically, TimeGraph adopts an *event-driven* scheduler model that synchronizes the GPU with the CPU in a responsive manner, using GPU-to-CPU interrupts, to schedule non-preemptive GPU commands for the asynchronously-operating GPU. Under this event-driven model, TimeGraph supports two scheduling policies to *prioritize* tasks on the GPU, which address the trade-off between response times and throughput. TimeGraph also employs two resource reservation policies to *isolate* tasks on the GPU, which provide different levels of quality of service (QoS) at the expense of different levels of overhead. To the best of our knowledge, this is the first work that enables GPU applications to be prioritized and isolated in real-time multi-tasking environments.

Organization: The rest of this paper is organized as follows. Section 2 introduces our system model, including the scope and limitations of TimeGraph. Section 3 provides the system architecture of TimeGraph. Section 4 and Section 5 describe the design and implementation of TimeGraph GPU scheduling and reservation mechanisms respectively. In Section 6, the performance of TimeGraph is evaluated, and its capabilities are demonstrated. Related work is discussed in Section 7. Our concluding remarks are provided in Section 8.

2 System Model

Scope and Limitations: We assume a system composed of a generic multi-core CPU and an on-board GPU. We do not manipulate any GPU-internal units, and hence GPU commands are not preempted once they are submitted to the GPU. TimeGraph is independent of libraries, compilers, and runtime engines. The principles of TimeGraph are therefore applicable for different GPU architectures (e.g., NVIDIA Fermi/Tesla and ATI Stream) and programming frameworks (e.g., OpenGL, OpenCL, CUDA, and HMPP). Currently, TimeGraph is designed and implemented for Nouveau [7] available in the Gallium3D [15] OpenGL software stack, which is also planned to support OpenCL. Moreover, TimeGraph has been ported to the PSCNV open-source driver [22] packaged in the PathScale ENZO suite [21], which supports CUDA and HMPP. This paper is, however, focused on OpenGL workloads, given the currently-available set of open-source solutions: Nouveau and Gallium3D.

Driver Model: TimeGraph is part of the device driver, which is an interface for user-space programs to submit GPU commands to the GPU. We assume that the device driver is designed based on the *Direct Rendering Infrastructure* (*DRI*) [14] model that is adopted in most UNIX-like operating systems, as part of the X Window System. Under the DRI model, user-space programs are allowed to access the GPU directly to render frames without using windowing protocols, while they still use the windowing server to blit the rendered frames to the screen. GPGPU frameworks require no such windowing procedures, and hence their model is more simplified.

In order to submit GPU commands to the GPU, user-space programs must be allocated GPU *channels*, which conceptually represent separate address spaces on the GPU. For instance, the NVIDIA Fermi and Tesla architectures support 128 channels. Our GPU command submission model for each channel is shown in Figure 2.

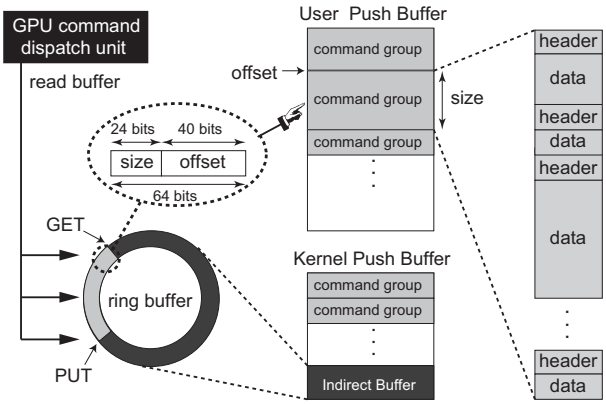


Figure 2: GPU command submission model.

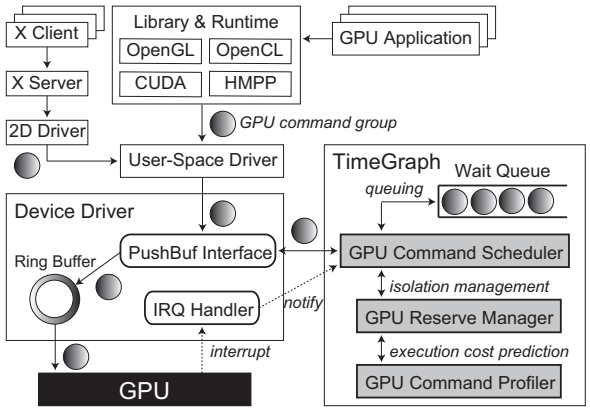


Figure 3: TimeGraph system architecture.

Each channel uses two types of kernel-space buffers: *User Push Buffer* and *Kernel Push Buffer*. The User Push Buffer is mapped on to the address space of the corresponding task, where GPU commands are pushed from the user space. GPU commands are usually *grouped* as *non-preemptive* regions to match user-space atomicity assumptions. The Kernel Push Buffer, meanwhile, is used for kernel primitives, such as host-device synchronization, GPU initialization, and GPU mode setting.

While user-space programs push GPU commands into the User Push Buffer, they also write *packets*, each of which is a (*size* and *address*) tuple to locate a certain GPU command group, into a specific ring buffer part of the Kernel Push Buffer, called *Indirect Buffer*. The driver configures the command dispatch unit on the GPU to read the buffer for command submission. This ring buffer is controlled by `GET` and `PUT` pointers. The pointers start from the same place. Every time packets are written to the buffer, the driver moves the `PUT` pointer to the tail of the packets, and sends a signal to the GPU command dispatch unit to download the GPU command groups located by the packets between the `GET` and `PUT` pointers. The `GET` pointer is then automatically updated to the same place as the `PUT` pointer. Once these GPU command groups are submitted to the GPU, the driver does not manage them any longer, and continues to submit the next set of GPU command groups, if any. Thus, this Indirect Buffer plays a role of a command queue.

Each GPU command group may include multiple GPU commands. Each GPU command is composed of the header and data. The header contains *methods* and the data size, while the data contains the values being passed to the methods. Methods represent GPU instructions, some of which are shared between compute and graphics, and others are specific for each. We assume that the device driver does not preempt on-the-fly GPU command groups, once they are offloaded on to the GPU. GPU command execution is out-of-order within the same GPU channel. The GPU channels are switched automatically by the GPU engines.

Our driver model described above is based on *Direct Rendering Manager* (DRM) [6], and especially target the NVIDIA Fermi and Tesla architectures, but can also be used for other architectures with minor modification.

3 TimeGraph System Architecture

The architecture of TimeGraph and its interaction with the rest of the software stack is illustrated in Figure 3. No modification is required for user-space programs, and GPU command groups can be generated through existing software frameworks. However, TimeGraph needs to communicate with a specific interface, called PushBuf, in the device driver space. The PushBuf interface enables the user space to submit GPU command groups stored in the User Push Buffer. TimeGraph uses this PushBuf interface to queue GPU command groups. It also uses the IRQ handler prepared for GPU-to-CPU interrupts to dispatch the next available GPU command groups.

TimeGraph is composed of *GPU command scheduler*, *GPU reserve manager*, and *GPU command profiler*. The GPU command scheduler queues and dispatches GPU command groups based on task priorities. It also coordinates with the GPU reserve manager to account and enforce GPU execution times of tasks. The GPU command profiler supports prediction of GPU command execution costs to avoid overruns out of reservation. There are two scheduling policies supported to address the trade-off between response times and throughput:

- Predictable-Response-Time (PRT): This policy minimizes priority inversion on the GPU to provide predictable response times based on priorities.
- High-Throughput (HT): This policy increases total throughput, allowing additional priority inversion.

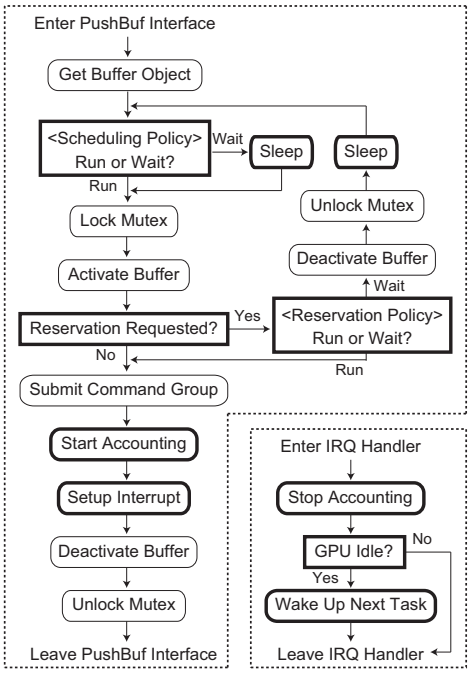


Figure 4: Diagram of the PushBuf interface and the IRQ handler with the TimeGraph scheme.

It also supports two GPU reservation policies that address the trade-off between isolation and throughput:

- Posterior Enforcement (PE): This policy enforces GPU resource usage after GPU command groups are completed without sacrificing throughput.
- Apriori Enforcement (AE): This policy enforces GPU resource usage before GPU command groups are submitted using prediction of GPU execution costs at the expense of additional overhead.

In order to unify multiple tasks into a single reserve, the TimeGraph reservation mechanism provides the Shared reservation mode. Particularly, TimeGraph creates a special Shared reserve instance with the PE policy when loaded, called Background, which serves all GPU-accelerated tasks that do not belong to any specific reserves. The detailed design and implementation for GPU scheduling and GPU reservation will be described in Section 4 and Section 5 respectively.

Figure 4 shows a high-level diagram of the PushBuf interface and the IRQ handler, where modifications introduced by TimeGraph are highlighted by bold frames. This diagram is based on the Nouveau implementation, but most GPU drivers should have similar control flows.

The PushBuf interface first acquires the buffer object associated with the incoming GPU command group. It then applies the scheduling policy to determine whether this GPU command group can execute on the GPU. If it should not be dispatched immediately, the corresponding task goes to sleep. Else, the User Push Buffer object is activated for command submission with the mutex lock to ensure the GPU command group to be located in the place accessible from the GPU, though the necessity of this procedure depends on driver implementation. TimeGraph next checks if GPU reservation is requested for this task. If so, it applies the reservation policy to verify the GPU resource usage of this task. If it overruns, TimeGraph winds up buffer activation, and suspends this task until its resource budget becomes available. This task will be rescheduled later when it is waken up, since some higher-priority tasks may arrive by then. Finally, if the GPU command group is qualified by the scheduling and reservation policies, it is submitted to the GPU. As the reservation policies need to track GPU resource usage, TimeGraph starts accounting for the GPU execution time of this task. It then configures the GPU command group to generate an interrupt to the CPU upon completion so that TimeGraph can dispatch the next GPU command group. After deactivating the buffer and unlocking the mutex, the PushBuf interface returns.

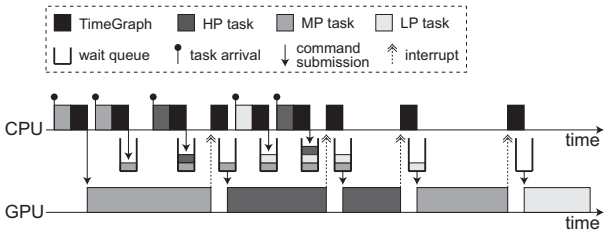
The IRQ handler receives an interrupt notifying the completion of the current GPU command group, where TimeGraph stops accounting for the GPU execution time, and wakes up the next task to execute on the GPU based on the scheduling policy, if the GPU is idle.

Specification: System designers may use a *specification* primitive to activate the TimeGraph functionality, which is inspired by the Redline system [31]. For each application, system designers can specify the scheduling parameters as: `<name:sched:resv:prio:C:T>`, where `name` is the application name, `sched` is its scheduling policy, `resv` is its reservation policy, `prio` is its priority, and a set of `C` and `T` represents that the application task is allowed to execute on the GPU for `C` microseconds every `T` microseconds. The specification is a text file (`/etc/timegraph.spec`), and TimeGraph reads it every time a new GPU channel is allocated to a task. If there is a matching entry based on the application name associated with the task, the specification is applied to the task. Otherwise, the task is assigned the lowest GPU priority and the Background reserve.

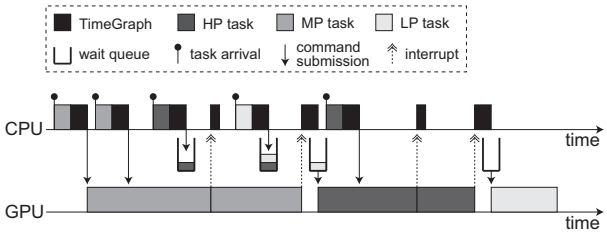
Priority Assignment: While system designers may assign static GPU priorities in their specification, TimeGraph also supports automatic GPU priority assignment (AGPA), which is enabled by using a wild-card "`*`" entry in the `prio` field. TimeGraph provides a user-space daemon executing periodically to identify the task with the foreground window through a window programming interface, such as the `_NET_ACTIVE_WINDOW` and the `_NET_WM_PID` properties in the X Window System. TimeGraph receives the foreground task information via a system call, and assigns the highest priority to this

task among those running under the AGPA mechanism. These tasks execute at the *default* static GPU priority level. Hence, different tasks can be prioritized over them by assigning higher static GPU priorities. AGPA is, however, not available if the above window programming interface is not supported. TimeGraph instead provides another user-space tool for system designers to assign priorities. For instance, designers can provide an optimal priority assignment based on reserve periods [13], as widely adopted in real-time systems.

Admission Control: In order to achieve predictable services in overloaded situations, TimeGraph provides an admission control scheme that forces the new reserve to be a background reserve so that currently active reserves continue to execute in a predictable manner. TimeGraph provides a simple interface where designers specify the limit of total GPU resource usage by 0-100% in a text file (/etc/timegraph.ac). The amount of limit is computed by a traditional resource-reservation model based on C and T of each reserve [26].



(a) PRT scheduling.



(b) HT scheduling.

Figure 5: Example of GPU scheduling in TimeGraph.

4 TimeGraph GPU Scheduling

The goal of the GPU command scheduler is to *queue* and *dispatch* non-preemptive GPU command groups in accordance with task priorities. To this end, TimeGraph contains a *wait queue* to stall tasks. It also manages a *GPU-online list*, a list of pointers to the GPU command groups currently executing on the GPU.

The GPU-online list is used to check if there are currently-executing GPU command groups, when a GPU command group enters into the PushBuf interface. If the list is empty, the corresponding task is inserted into it, and the GPU command group is submitted to the GPU. Else, the task is inserted into the wait queue to be scheduled. The scheduling policies supported by TimeGraph will be presented in Section 4.1.

Management of the GPU-online list requires the information about when GPU command groups complete. TimeGraph adopts an event-driven model that uses GPU-to-CPU interrupts to notify the completion of each GPU command group, rather than a tick-driven model adopted in the previous work [1, 5]. Upon every interrupt, the corresponding GPU command group is removed from the GPU-online list. Our GPU-to-CPU interrupt setting and handling mechanisms will be described in Section 4.2.

4.1 Scheduling Policies

TimeGraph supports two GPU scheduling policies. The Predictable-Response-Time (PRT) policy encourages such tasks that should behave on a timely basis without affecting important tasks. This policy is predictable in a sense that GPU command groups are scheduled based on task priorities to make high-priority tasks responsive on the GPU. The High-Throughput (HT) policy, on the other hand, is suitable for such tasks that should execute as fast as possible. There is a trade-off that the PRT policy prevents tasks from interference at the expense of throughput, while the HT policy achieves high throughput for one task but may block others. For instance, desktop-widget, browser-plugin, and video-player tasks are desired to use the PRT policy, while 3-D game and interactive 3-D interfacing tasks can use the HT policy.

PRT Scheduling: The PRT policy forces any GPU command groups to wait for the completion of the preceding GPU command group, if any. Specifically, a new GPU command group arriving at the device driver can be submitted to the GPU immediately, if the GPU-online list is empty. Else, the corresponding task must sleep in the wait queue. The highest-priority task in the wait queue, if any, is waken up upon every interrupt from the GPU.

Figure 5 (a) indicates how three tasks with different priorities, high-priority, medium-priority (MP), and low-priority (LP), are scheduled on the GPU under the PRT policy. When the MP task arrives, its GPU command group can execute on the GPU, since no GPU command groups are executing. Given that the GPU and CPU operate asynchronously, the MP task can arrive again while its previous GPU command group is executing. However, the MP task is queued this time, because the GPU is not idle, according to the PRT policy. Even the next HP task is also queued due to the same reason, since further higher-priority tasks may arrive soon. The specific set of GPU commands appended at the end of every GPU command group by TimeGraph generates an interrupt to the CPU, and the TimeGraph scheduler is invoked ac-

cordingly to wake up the highest-priority task in the wait queue. Hence, the HP task is next chosen to execute on the GPU rather than the MP task. In this manner, the next instance of the LP task and the second instance of the HP task are scheduled in accordance with their priorities.

Given that the arrival times of GPU command groups are not known a priori, and each GPU command group is non-preemptive, we believe that the PRT policy is the best possible approach to provide predictable response times. However, it inevitably incurs overhead to make a scheduling decision at every GPU command group boundary, as shown in Figure 5 (a).

HT Scheduling: The HT policy reduces this scheduling overhead, compromising predictable response times a bit. It allows GPU command groups to be submitted to the GPU immediately, if (i) the currently-executing GPU command group was submitted by the same task, and (ii) no higher-priority tasks are ready in the wait queue. Otherwise, they must suspend in the same manner as the PRT policy. Upon an interrupt, the highest-priority task in the wait queue is waken up, *only when* the GPU-online list is empty (the GPU is idle).

Figure 5 (b) depicts how the same set of GPU command groups used in Figure 5 (a) is scheduled under the HT policy. Unlike the PRT policy, the second instance of the MP task can submit its GPU command group immediately, because the currently-executing GPU command group was issued by itself. These two GPU command groups of the MP task can execute successively without producing the idle time. The same is true for the two GPU command groups of the HP task. Thus, the HT policy is more for throughput-oriented tasks, but the HP task is blocked by the MP task for a longer internal. This is a trade-off, and if priority inversion is critical, the PRT policy is more appropriate.

4.2 Interrupt Setting and Handling

In order to provide an event-driven model, TimeGraph configures the GPU to generate an interrupt to the CPU upon the completion of each GPU command group. The scheduling point is thus made at every GPU command group boundary. We now describe how the interrupt is generated. For simplicity of description, we here focus on the NVIDIA GPU architecture.

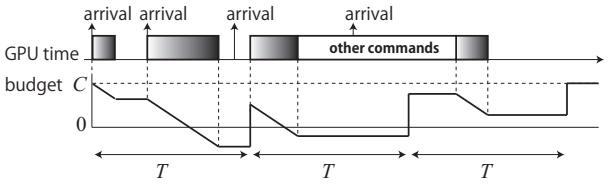
Completion Notifier: The NVIDIA GPU provides the `NOTIFY` command to generate an interrupt from the GPU to the CPU. TimeGraph puts this command at the end of each GPU command group. However, the interrupt is not launched immediately when the `NOTIFY` command is operated but when the next command is dispatched. TimeGraph therefore adds the `NOP` command after the `NOTIFY` command, as a dummy command. We also need to consider that GPU commands execute out of order on the GPU. If the `NOTIFY` command is operated before all commands in the original GPU command group are operated, the generated interrupt is not timely at all. TimeGraph hence adds the `SERIALIZE` command right before the `NOTIFY` command, which forces the GPU to stall until all on-the-fly commands complete. There is no need to add another piece of the `SERIALIZE` command after the `NOTIFY` command, since we know that no tasks other than the current task can use the GPU until TimeGraph is invoked upon the interrupt.

Interrupt Association: All interrupts from the GPU caught in the IRQ handler are relayed to TimeGraph. When TimeGraph receives an interrupt, it first references the head of the GPU-online list to obtain the task information associated with the corresponding GPU command group. TimeGraph next needs to verify whether this interrupt is truly generated by the commands that TimeGraph inserted into at the end of the GPU command group, given that user-space programs may also use the `NOTIFY` command. In order to recognize the right interrupt, TimeGraph further adds the `SET_REF` command before the `SERIALIZE` command, which instructs the GPU to write a specified sequence number to a particular GPU register. This number is identical for each task, and is simply incremented by TimeGraph. TimeGraph reads this GPU register when an interrupt is received. If the register value is less than the expected sequence number associated with the corresponding GPU command group, this interrupt should be ignored, since it must have been caused by someone else before the `SET_REF` command. Another piece of the `SERIALIZE` command also needs to be added before the `SET_REF` command to ensure in-order command execution. As a consequence, TimeGraph inserts the following commands at the end of each GPU command group: `SERIALIZE`, `SET_REF`, `SERIALIZE`, `NOTIFY`, `NOP`.

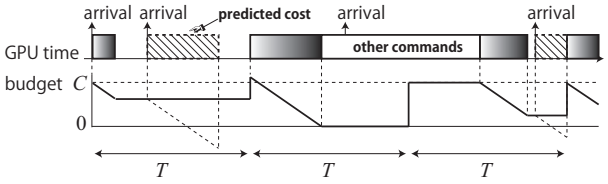
Task Wake-Up: Once the interrupt is verified, TimeGraph removes the GPU command group at the head of the GPU-online list. If the corresponding task is scheduled under the PRT policy, TimeGraph wakes up the highest-priority task in the wait queue, and inserts its GPU command group into the GPU-online list. If the task is assigned the HT policy, meanwhile, TimeGraph wakes up the highest-priority task in the same manner as the PRT policy, *only when* the GPU-online list is empty.

5 TimeGraph GPU Reservation

TimeGraph provides GPU reservation mechanisms to regulate GPU resource usage for tasks scheduled under the PRT policy. Each task is assigned a *reserve* that is represented by capacity C and period T. Budget e is the amount of time that a task is entitled for execution. TimeGraph uses a popular rule for budget consumption and



(a) PE reservation.



(b) AE reservation.

Figure 6: Example of GPU reservation in TimeGraph.

replenishment used in real-time systems [20, 26]. Specifically, the budget is decreased by the amount of time consumed on the GPU, and is replenished by at most capacity C once every period T. However, we need different reservation policies than previous work due to the asynchronous and non-preemptive nature of GPU processing, as we will describe in Section 5.1. Our GPU resource accounting and enforcement mechanisms will be described in Section 5.2. TimeGraph further supports prediction of GPU execution costs for strict isolation. Section 5.3 will describe our approach to GPU execution cost prediction.

5.1 Reservation Policies

TimeGraph supports two GPU reservation policies. The Posterior Enforcement (PE) policy is aimed for lightweight reservation, allowing tasks to overrun out of their reserves to an extent. The Apriori Enforcement (AE) policy reduces reserve overruns by predicting GPU execution costs a priori at the expense of additional overhead. We recommend that the PE policy be primarily used when isolation is required, and the AE policy be used only if extremely time-critical applications are concurrently executed on the GPU.

PE Reservation: The PE policy permits GPU command groups to be submitted to the GPU, if their budget is greater than zero. Else, the task goes to sleep until the budget is replenished. The budget can be negative, when the task overruns out of reservation. The overrun penalty is, however, imposed on the next budget replenishment. The budget for the next period is therefore given by $e = \min(C, e + C)$.

Figure 6 (a) shows how four GPU command groups of the same task are enforced under the PE policy. The budget is initialized to C. When the second GPU command group completes, the budget is negative. Hence, the third GPU command group must wait for the budget to be replenished, even though the GPU remains idle. Since GPU reservation is available under the PRT policy, the fourth GPU command group is blocked even though the budget is greater than zero, since another GPU command group is currently executing.

AE Reservation: For each GPU command group submission, the AE policy first predicts a GPU execution cost x. The GPU command group can be submitted to the GPU, only if the predicted cost is no greater than the budget. Else, the task goes to sleep until the budget is replenished. The next replenishment amount depends on the predicted cost x and the currently-remaining budget e. If the predicted cost x is no greater than the capacity C, the budget for the next period is bounded by $e = C$ to avoid transient overload. Else, it is set to $e = \min\{x, e + C\}$. The task can be waken up only when $e \geq x$.

Figure 6 (b) depicts how the same set of four GPU command groups used in Figure 6 (a) is controlled under the AE policy. For simplicity of description, we assume for now that prediction of GPU execution costs is perfectly accurate, and Section 5.3 will describe how to practically predict GPU execution costs. Unlike the PE policy, the second GPU command group is not submitted to the GPU, as its budget is less than the predicted cost, but is submitted later when the budget is replenished to be $e = \min\{x, e + C\} > x$. The fourth GPU command group also needs to wait until the budget is sufficiently replenished. However, unlike the second GPU command group, the replenished budget is bounded by C, since $x < C$. This avoids transient overload.

Shared Reservation: TimeGraph allows multiple tasks to share a single reserve under the Shared mode. When some task creates a Shared reserve, other tasks can join it. The Shared mode can be used together with both the PE and AE policies. The Shared mode is useful when users want to cap the GPU resource usage of multiple tasks to a certain range. There is no need to adjust the capacity and period for each task. It can also reduce the overhead of reservation, since it only needs to manage one reserve for multiple tasks.

5.2 Accounting and Enforcement

GPU execution times are accounted in the PushBuf interface and the IRQ handler as illustrated in Figure 4. TimeGraph saves CPU timestamps when GPU command groups start and complete. Specifically, when each GPU command group is qualified to be submitted to the GPU, TimeGraph records the current CPU time as its *start time* in the PushBuf interface, and at some later point of time when TimeGraph is notified of the completion of this GPU command group, the current CPU time is recorded as its *finish time* in the IRQ handler. The difference be-

tween the start time and the finish time is accounted for as the execution time of this GPU command group, and is subtracted from the budget.

Enforcement works differently for the PE and the AE policies. In the PushBuf interface, the AE policy predicts the execution cost x of each GPU command group based on the idea presented in Section 5.3, while the PE policy always assumes $x = 0$. Then, both policies compare the budget e and the cost x. Only if $e > x$ is satisfied, the GPU command group can be submitted to the GPU. Otherwise, the corresponding task is suspended until the budget is replenished. It should be noted that this enforcement mechanism is very different from traditional CPU reservation mechanisms [20, 26] that use timers or ticks to suspend tasks, since GPU command groups are non-preemptive, and hence we need to perform enforcement at GPU command group boundary. TimeGraph however still uses timers to replenish the budget periodically. Every time the budget is replenished, it compares e and x again. If $e > x$ is satisfied, the task is waken up, but it needs to be rescheduled, as illustrated in Figure 4.

5.3 Command Profiling

TimeGraph contains the GPU command profiler to predict GPU execution costs for AE reservation. Each GPU command is composed of the header and data, as shown in Figure 2. We hence parse the methods and the data sizes from the headers.

We now explain how to predict GPU execution costs from these pieces of information. GPU applications tend to repeatedly create GPU command groups with the same methods and data sizes, since they use the same set of API functions, e.g., OpenGL, and each function likely generates the same sequence of GPU commands in terms of methods and data sizes, while data values are quite variant. Given that GPU execution costs depend highly on methods and data sizes, but not on data values, we propose a history-based prediction approach.

TimeGraph manages a history table to record the GPU command group information. Each record consists of a *GPU command group matrix* and the average GPU execution cost associated to this matrix. The row and the column of the matrix contain the methods and their data sizes respectively. TimeGraph also attaches a flag to each GPU command group, indicating if it hits some record. When the methods and the data sizes of the GPU command group are obtained from the remapped User Push Buffer, TimeGraph looks at the history table. If there exists a record that contains exactly the same GPU command group matrix, i.e., the same set of methods and data sizes, it uses the average GPU execution cost stored in this record, and the flag is set. Otherwise, the flag is cleared, and TimeGraph uses the worst-case GPU execution cost among all the records. Upon the completion of the GPU command group, TimeGraph references the flag attached to the corresponding task. If the flag is set, it updates the average GPU execution cost of the record with the actual execution time of this GPU command group. Otherwise, it inserts a new record where the matrix has the methods and the data size of this GPU command group, and the average GPU execution time is initialized with its actual execution time. The size of the history table is configurable by designers. If the total number of the records exceeds the table size, the least-recently-used (LRU) record is removed.

Preemption Impact: Even the same GPU command group may consume very different GPU execution times. For example, if reusable texture data is cached, graphics operation is much faster. We realize that when the GPU contexts (channels) are switched, GPU execution times can vary. Hence, TimeGraph verifies GPU context switches at every scheduling point. If the context is switched, TimeGraph will not update the average GPU execution cost, since the context switch may have affected the actual GPU execution time. Instead, it saves the difference between the actual GPU execution time and the average GPU execution cost as the *preemption impact*. TimeGraph keeps updating the average preemption impact. A single preemption cost is measured beforehand when TimeGraph is loaded. The preemption impact is then added to the predicted cost.

6 Evaluation

We now provide a detailed quantitative evaluation of TimeGraph on the NVIDIA GeForce 9800 GT graphics card with the default frequency and 1 GB of video memory. Our underlying platform is the Linux 2.6.35 kernel running on the Intel Xeon E5504 CPU and 4 GB of main memory. While our evaluation and discussion are focused on this graphics card, Similar performance benefits from TimeGraph have also been observed with different graphics cards viz, GeForce GTX 285 and GTX 480.

As primary 3-D graphics benchmarks, we use the Phoronix Test Suite [24] that executes the OpenGL 3-D games, *OpenArena*, *World of Padman*, *Urban Terror*, and *Unreal Tournament 2004 (UT2004)*, in the demo mode based on the test profile, producing various GPU-intensive workloads. We also use *MPlayer* as a periodic workload. In addition, the Gallium3D *Engine* demo program is used as a regularly-behaved workload, and the Gallium3D *Clearspd* demo program that exploits a GPU command *bomb* is used as a misbehaving workload. Furthermore, we use *SPECviewperf 11* [28] to evaluate the throughput of different GPU scheduling models. The screen resolution is set to 1280×1024. The scheduling parameters are loaded from the pre-

configured TimeGraph specification file. The maximum number of records in the history table for GPU execution cost prediction is set to 100.

6.1 Prioritization and Isolation

We first evaluate the prioritization and isolation properties achieved by TimeGraph. As described in Section 3, TimeGraph automatically assigns priorities. CPU `nice` priorities are always effective, while GPU priorities are effective only when TimeGraph is activated. The priority level is aligned between the GPU and CPU. We use the PRT policy for the X server to prevent it from affecting primary applications, but it is scheduled by the highest GPU/CPU priority, since it should still be responsive to blit the rendered frames to the screen.

Coarse-grained Performance: Figure 7 shows the performance of the 3-D games, while the Engine widget is concurrently sharing the GPU. We use the HT policy for the 3-D games, while the Engine widget is assigned the PRT policy under TimeGraph. As shown in Figure 7, TimeGraph improves the performance of the 3-D games by about 11% for OpenArena, 27% for World of Padman, 22% for Urban Terror, and 2% for UT2004, with GPU priority support. Further performance isolation is obtained by GPU reservation, capping the GPU resource usage of the Engine widget. Our experiment assigns the Engine widget a reserve of $2.5ms$ every $25ms$ to retain GPU resource usage at 10%. As compared to the case without GPU reservation support, the performance of the 3-D games is improved by $2 \sim 21\%$ under PE reservation, and by $4 \sim 36\%$ under AE reservation. Thus, the AE policy provides better performance for the 3-D games at the expense of more conservative scheduling of the Engine widget with prediction.

Figure 8 presents the results from a setup similar to the above experiments, where the Clearspd bomb generates heavily-competing workload instead of the Engine widget. The performance benefit resulting from assigning higher GPU priorities to the games under the HT policy is clearer in this setup. Even without GPU reservation support, TimeGraph enables the 3-D games to run about $3 \sim 6$ times faster than the vanilla Nouveau driver, though they still face a performance loss of about $24 \sim 52\%$ as compared to the previous setup where the Engine widget contends with the 3-D games. Regulating the GPU resource usage of the Clearspd bomb through GPU reservation limits this performance loss to be within 3%. Particularly, the AE policy yields improvements of up to 5% over the PE policy.

Extreme Workloads: In order to evaluate the capabilities of TimeGraph in the face of extreme workloads, we execute the 3-D games with five instances of the Clearspd bomb. In this case, the cap of each individual reserve is correspondingly decreased to $0.5ms$ every $25ms$ so that the total cap of the five Clearspd-bomb tasks is aligned with $2.5ms$ every $25ms$. As here are multiple Clearspd-bomb tasks, we evaluate an additional setup where a single PE reserve of $2.5ms$ every $25ms$ runs with the Shared reservation mode. As shown in Figure 9, the 3-D games are nearly unresponsive without TimeGraph support due to the scaled-up GPU workload, whereas TimeGraph can isolate the performance of the 3-D games even under such an extreme circumstance. In fact, the performance impact is reduced to $7 \sim 20\%$ by using GPU priorities, and leveraging GPU reservation re-

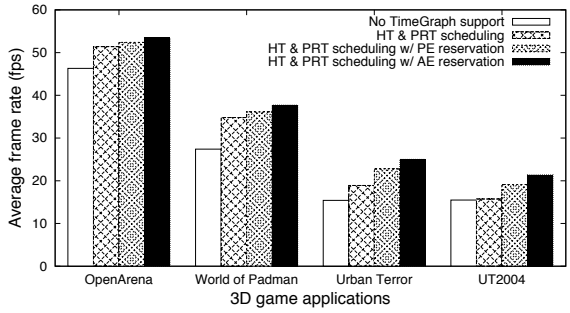


Figure 7: Performance of the 3-D games competing with a single instance of the Engine widget.

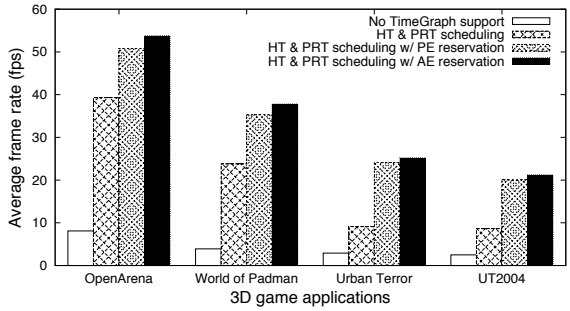


Figure 8: Performance of the 3-D games competing with a single instance of the Clearspd bomb.

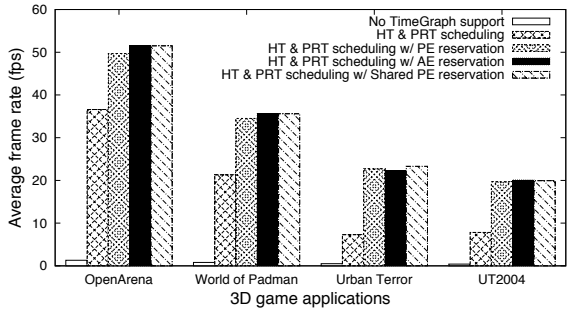


Figure 9: Performance of the 3-D games competing with five instances of the Clearspd bomb.

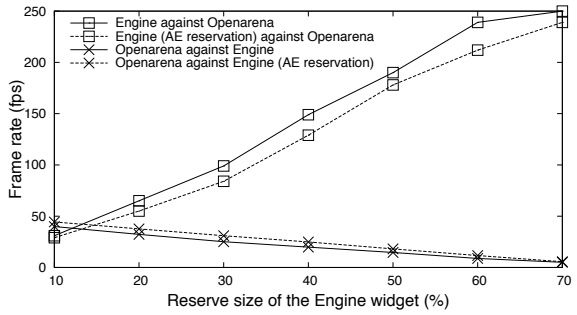


Figure 10: Performance regulation by GPU reservation for the 3-D game and the 3-D widget.

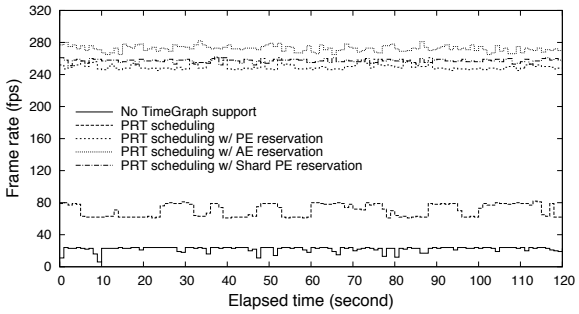
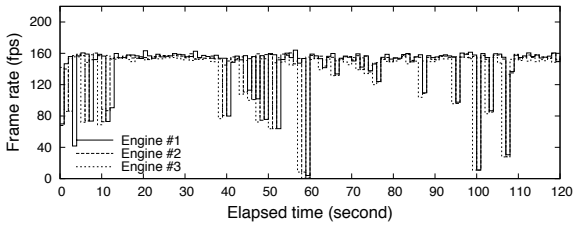
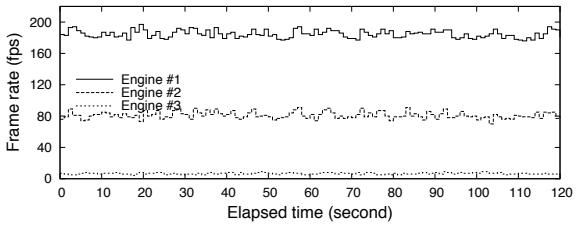


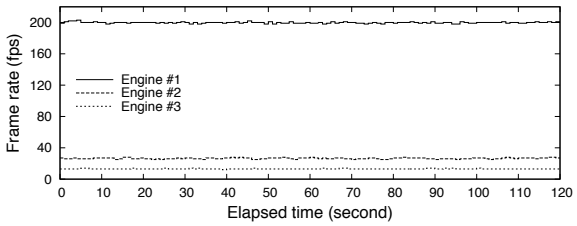
Figure 11: Performance of the Engine widget competing with five instances of the Clearspd bomb.



(a) No TimeGraph support.



(b) PRT scheduling.



(c) PRT scheduling and PE reservation.

Figure 12: Interference among three widget instances.

sults in nearly no performance loss, similar to results in Figure 8. The Shared reservation mode also provides slightly better performance with PE reserves.

Performance Regulation: We next demonstrate the effectiveness of TimeGraph in regulating the frame-rate for each task by changing the size of GPU reserve. Figure 10 shows the performance of the OpenArena game and the Engine widget contending with each other. The solid lines indicate a setup where the PE policy is assigned for both the applications, while the dotted lines indicate a setup where the AE policy is assigned for the Engine widget instead. GPU reservation is configured so that the total GPU resource usage of the two applications is capped at 90%, and the remaining 10% is available for the X server. Assigning the AE policy for the Engine widget slightly improves the performance of the OpenArena game, while it brings a performance penalty for the Engine widget itself due to the overhead for prediction of GPU execution costs. In either case, however, TimeGraph successfully regulates the frame-rate in accordance with the size of GPU reserve. In this experiment, we conclude that it is desirable to assign a GPU reserve for the OpenArena game with $C/T = 60 \sim 80\%$ and that for the Engine widget with $C/T = 10 \sim 30\%$, given that this configuration provides both the applications with an acceptable frame-rate over 25 fps.

Fine-grained Performance: The 3-D games demonstrate highly variable frame-rate workloads, while 3-D widgets often exhibit nearly constant frame-rates. In order to study the behavior of TimeGraph on both these two categories of applications, we look at the variability of frame-rate with time for the Engine widget contending with five instances of the Clearspd bomb, as shown in Figure 11. The total GPU resource usage of the Clearspd-bomb tasks is capped at $2.5ms$ every $25ms$ through GPU reservation, and a higher priority is given to the Engine widget. These results show that GPU reservation can provide stable frame-rates on a time for the Engine widget. Since the Engine widget is not as GPU-intensive as the 3-D games, it is affected more by the Clearspd bomb making the GPU overloaded, when GPU reservation is not applied. The benefits of GPU reservation are therefore more clearly observed.

Interference Issues: We now evaluate the interference among regularly-behaved concurrent 3-D widgets. Figure 12 (a) shows a chaotic behavior arising from executing three instances of the Engine widget concurrently, with different CPU priorities but without TimeGraph support. Although the Engine widget by itself is a very regular workload, when competing with more instances of itself, the GPU resource usage exhibits high variabil-

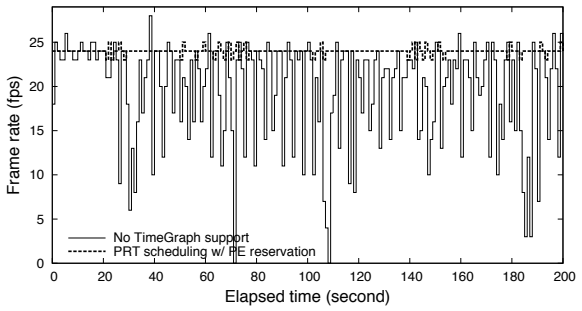


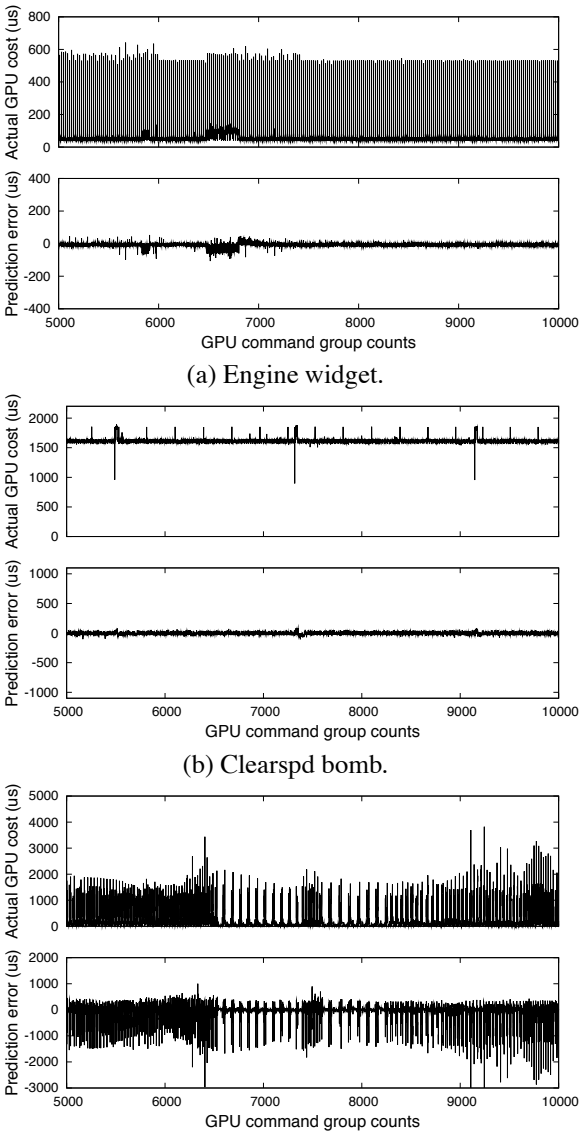
Figure 13: Performance of MPlayer competing with five instances of the Clearspd bomb.

ity and unpredictability. Figure 12 (b) illustrates the improved behavior under TimeGraph using the PRT policy, where we assign the high, the medium, and the low GPU priorities for *Engine #1*, *Engine #2*, and *Engine #3* respectively, using the user-space tool presented in Section 3. TimeGraph successfully provides predictable response times for the three tasks according based on their priorities. Further performance isolation can be achieved by GPU reservation, exploiting different sizes of GPU reserves: (i) $15ms$ every $25ms$ to *Engine #1*, (ii) $5ms$ every $50ms$ to *Engine #2*, and (iii) $5ms$ every $100ms$ to *Engine #3*, as shown in Figure 12 (c). The PE policy is used here. Since the Engine widget has a non-trivial dependence on the CPU, the absolute performance is lower than expected for smaller reserves.

Periodic Behavior: For evaluating the impact on applications with periodic activity, we execute MPlayer in the foreground when five instances of the Clearspd bomb contend for the GPU. We use an H264-compressed video, with a frame size of 1920×800 and a frame rate of 24 fps, which uses x-video acceleration on the GPU. As shown in Figure 13, the video playback experience is significantly disturbed without TimeGraph support. When TimeGraph assigns a PE reserve of $10ms$ every $40ms$ for MPlayer, and a PE reserve of $5ms$ every $40ms$ for the Clearspd bomb tasks in the Shared reservation mode, the playback experience is significantly improved. It closely follows the ideal frame-rate of 24 fps for video playback. This illustrates the benefits of TimeGraph for interactivity, where performance isolation plays a vital role in determining user experience.

6.2 GPU Execution Cost Prediction

We now evaluate the history-based prediction of GPU execution costs for realizing GPU reservation with the AE policy. The effectiveness of AE reservation relies highly on GPU execution cost prediction. Hence, it is important to identify the types of applications for which we



(a) Engine widget.

(b) Clearspd bomb.

(c) OpenArena game.

Figure 14: Errors for GPU execution cost prediction.

can predict GPU execution costs more precisely. Figure 14 shows both actual GPU execution costs and prediction errors for the 3-D graphics applications used in earlier experiments: Engine, Clearspd, and OpenArena. Since these applications issue a very large number of GPU command groups, we focus on a snapshot between the 5000th and the 10000th GPU command groups of the applications. In the following discussion, positive errors denote pessimistic predictions. Figure 14 (a) shows that the history-based prediction approach employed by TimeGraph performs within a 15% error margin on the Engine widget that uses a reasonable number of methods. For the Clearspd bomb that issues a very limited set of methods, while producing extreme workloads, Time-

Graph can predict GPU execution costs within a 7% error margin as shown in Figure 14 (b). On the other hand, the results of GPU execution cost prediction under OpenArena, provided in Figure 14 (c), show that only about 65% of the observed GPU command groups have the predicted GPU execution costs within a 20% error margin. Such unpredictability arises from the inherently dynamic nature of complex computer graphics like abrupt scene changes. The actual penalty of misprediction is, however, suffered only once per reserve period, and is hence not expected to be significant for reserves with long periods.

In addition to the presented method, we have also explored static approaches using pre-configured values for predicted costs. Our experiments show that such static approaches perform worse than the presented dynamic approach, largely due to the dynamically changing and non-stationary nature of application workloads.

GPU execution cost prediction plays a vital role in real-time setups, where it is unacceptable for low-priority tasks to even cause the slightest interference to high-priority tasks. As the above experimental results show that our prediction approach tends to fail for complex interactive applications like OpenArena. However, we expect the structure of real-time applications to be less dynamic and more regular like the Engine and Clearspd tasks. GPU reservation with the AE policy for complex applications like OpenArena would require support from the application program itself, since their behavior is not easily predictable from historic execution results. Otherwise, the PE policy is desired for low overhead.

6.3 Overhead and Throughput

In order to quantify the performance overhead imposed by TimeGraph, we measure the standalone performance of the 3-D game benchmarks. Figure 15 shows that assigning the HT policy for both the games and the X server incurs about 4% performance overhead for the games. This small overhead is attributed to the fact that TimeGraph is still invoked upon every arrival and completion of GPU command group. It is interesting to see that assigning the PRT policy for the X server increases the overhead for the games up to about 10%, even though the games use the HT policy. As the X server is used to blit the rendered frames to the screen, it can lower the frame-rate of the game, if it is blocked by the game itself. On the other hand, assigning the PRT policy for both the X server and the game adds a non-trivial overhead of about $17 \sim 28\%$ largely due to queuing and dispatching all GPU command groups. This overhead is introduced by queuing delays and scheduling overheads, as TimeGraph needs to be invoked for submission of each GPU command group. We however conjecture that such overhead cost is inevitable for GPU scheduling at the device-driver level to shield important GPU applications from performance interference. If there is no TimeGraph support, the performance of high-priority GPU applications could significantly decrease in the presence of competing GPU workloads (as shown in Figure 1), which affects the system performance more than the maximum scheduling overhead of 28% introduced by TimeGraph. As a consequence, TimeGraph is evidently beneficial in real-time multi-tasking environments.

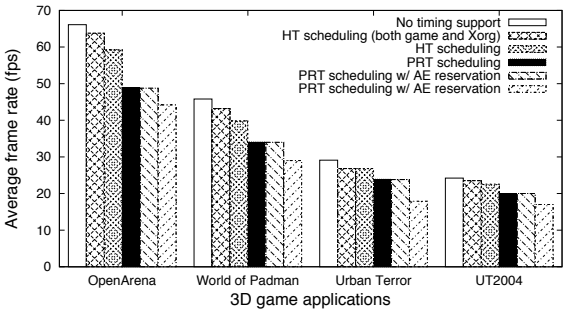


Figure 15: Performance overheads of TimeGraph.

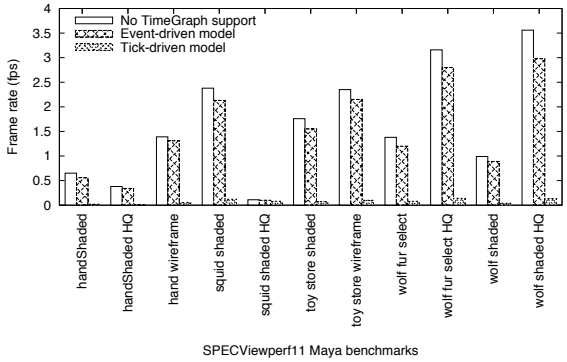


Figure 16: Throughput of event-driven and tick-driven schedulers in TimeGraph.

Finally, we compare the throughput of two conceivable GPU scheduling models: (i) the event-driven scheduler adopted in TimeGraph, and (ii) the tick-driven scheduler that was presented in the previous work, called GERM [1, 5]. The TimeGraph event-driven scheduler is invoked only when GPU command groups arrive and complete, while the GERM tick-driven scheduler is invoked at every tick, configured to be $1ms$ (Linux *jiffies*). Figure 16 shows the results of SPECviewperf benchmarking with the *Maya* viewset created for 3-D computer graphics, where the PRT policy is used for GPU scheduling. Surprisingly, the TimeGraph event-driven scheduler obtains about $15 \sim 30$ times better scores than the tick-driven scheduler for most test cases. According to our analysis, this difference arises from the fact that many

GPU command groups can arrive in a very short interval. The GERM tick-driven scheduler reads a particular GPU register every tick to verify if the current GPU command group has completed. Suppose that there are 30 GPU command groups with a total execution time of less than $1ms$. The tick-driven scheduler takes at least $30ms$ to compete these GPU command groups because the GPU register must be read 10 times, while the event-driven scheduler could complete them in $1ms$ as GPU-to-CPU interrupts are used. Hence, non-trivial overheads are imposed on the tick-driven scheduler.

7 Related Work

GPU Scheduling: The Graphics Engine Resource Manager (GERM) [1, 5] aims for GPU multi-tasking support similar to TimeGraph. The resource management concepts of TimeGraph and GERM are, however, fundamentally different. TimeGraph focuses on prioritization and isolation among competing GPU applications, while fairness is a primary concern for GERM. Since fair resource allocation cannot shield particular important tasks from interference in the face of extreme workloads, as reported in [31], TimeGraph addresses this problem for GPU applications through priority and reservation support. Approaches to synchronize the GPU with the CPU are also different between TimeGraph and GERM. TimeGraph is based on an event-driven model that uses GPU-to-CPU interrupts, whereas GERM adopts a tick-driven model that polls a particular GPU register. As demonstrated in Section 6.3, the tick-driven model can become unresponsive when many GPU commands arrive in a short interval, which could likely happen for graphics and compute-intensive workloads, while TimeGraph is responsive even in such cases. Hence, TimeGraph is more suitable for real-time applications. In addition, TimeGraph can *predict* GPU execution costs a priori, taking into account both methods and data sizes, while GERM *estimates* them posteriorly, using only data sizes. Since GPU execution costs are very dependent not only on data sizes but also on methods, we claim that TimeGraph computes GPU execution costs more precisely. However, additional computation overheads are required for prediction. TimeGraph therefore provides light-weight reservation with the PE policy without prediction to address this trade-off. Furthermore, TimeGraph falls inside the device driver, while GERM is spread across the device driver and user-space library. Hence, GERM could require major modifications for different runtime frameworks, e.g., OpenGL, OpenCL, CUDA, and HMPP.

The Windows Display Driver Model (WDDM) [25] is a GPU driver architecture for the Microsoft Windows. While it is proprietary, GPU priorities seem to be supported in our experience, but are not explicitly exposed to the user space as a first-class primitive. Apparently, there is no GPU reservation support. In fact, since NVIDIA shares more than 90% of code between Linux and Windows [23]. Therefore, it eventually suffers from the performance interference as demonstrated in Figure 1.

VMGL [11] supports virtualization in the OpenGL APIs for graphics applications running inside a Virtual Machine (VM). It passes graphics requests from guest OSes to a VMM host, but GPU resource management is left to the underlying device driver. The GPU-accelerated Virtual Machine (GViM) [8] virtualizes the GPU at the level of abstraction for GPGPU applications, such as the CUDA APIs. However, since the solution is 'above' the device driver layer, GPU resource management is coarse-grained and functionally limited. VMware's Virtual GPU [3] enables GPU virtualization at the I/O level. Hence, it operates faster and its usage is not limited to GPGPU applications. However, multi-tasking support with prioritization, isolation, or fairness is not supported. TimeGraph could coordinate with these GPU virtualization systems to provide predictable response times and isolation.

CPU Scheduling: TimeGraph shares the concept of priority and reservation, which has been well-studied by the real-time systems community [13, 26], but there is a fundamental difference from these traditional studies in that TimeGraph is designed to address an arbitrarily-arriving non-preemptive GPU execution model, whereas the real-time systems community has often considered a periodic preemptive CPU execution model. Several bandwidth-preserving approaches [12, 29, 30] for an arbitrarily-arriving model exist, but a non-preemptive model has not been much studied yet. The concept of priority and reservation has also been considered in the operating systems literature [4, 9, 10, 17, 20, 31]. Specifically, batch scheduling has a similar constraint to GPU scheduling in that non-preemptive regions disturb predictable responsiveness [27]. These previous work are, however, mainly focused on synchronous on-chip CPU architectures, whereas TimeGraph addresses those scheduling problems for asynchronous on-board GPU architectures where explicit synchronization between the GPU and CPU is required.

Disk Scheduling: Disk devices have similarity to GPUs in that they operate with non-preemptive regions off the chip. Disk scheduling for real-time and interactive systems [2, 16] therefore considered priority and reservation support for non-preemptive operation. However, the GPU is typically a coprocessor independent of the CPU, which has its own set of execution contexts, registers, and memory devices, while the disk is more dedicated to I/O. Hence, TimeGraph uses completely different mechanisms to realize prioritization and isolation than these previous work. In addition, TimeGraph needs to ad-

dress the trade-off between predictable response times and throughput since synchronizing the GPU and CPU incurs overhead, while disk I/O is originally synchronous with read and write operation.

8 Conclusions

This paper has presented TimeGraph, a GPU scheduler to support real-time multi-tasking environments. We developed the event-driven model to schedule GPU commands in a responsive manner. This model allowed us to propose two GPU scheduling policies, Predictable Response Time (PRT) and High Throughput (HT), which address the trade-off between response times and throughput. We also proposed two GPU reservation policies, Posterior Enforcement (PE) and the Apriori Enforcement (AE), which present an essential design knob for choosing the level of isolation and throughput. Our detailed evaluation demonstrated that TimeGraph can protect important GPU applications even in the face of extreme GPU workloads, while providing high-throughput, in real-time multi-tasking environments. TimeGraph is open-source software, and may be downloaded from our website at `http://rtml.ece.cmu.edu/projects/timegraph/`.

In future work, we will elaborate coordination of GPU and CPU resource management schemes to further consolidate prioritization and isolation capabilities for the entire system. We are also interested in coordination of video memory and system memory management schemes. Exploration of other models for GPU scheduling is another interesting direction of future work. For instance, modifying the current API to introduce non-blocking interfaces could improve throughput at the expense of modifications to legacy applications. Scheduling overhead and blocking time may also be reduced by implementing an real-time *satellite kernel* [18] on microcontrollers present in modern GPUs. Finally, we will tackle the problem of mapping application-level specifications, such as frame-rates, into priority and reservation properties at the operating-system level.

References

[1] BAUTIN, M., DWARAKINATH, A., AND CHIUEH, T. Graphics Engine Resource Management. In *Proc. MMCN* (2008).

[2] DIMITRIJEVIC, Z., RANGAWAMI, R., AND CHANG, E. Design and Implementation of Semi-preemptible IO. In *Proc. USENIX FAST* (2003).

[3] DOWTY, M., AND SUGEMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[4] DUDA, K., AND CHERITON, D. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. In *Proc. ACM SOSP* (1999), pp. 261–276.

[5] DWARAKINATH, A. A Fair-Share Scheduler for the Graphics Processing Unit. Master's thesis, Stony Brook University, 2008.

[6] FAITH, R. *The Direct Rendering Manager: Kernel Support for the Direct Rendering Infrastructure*. Precision Insight, Inc., 1999.

[7] FREEDESKTOP. Nouveau Open-Source Driver. `http://nouveau.freedesktop.org/`.

[8] GUPTA, V., GAVRILOVSKA, A., TOLIA, N., AND TALWAR, V. GViM: GPU-accelerated Virtual Machines. In *Proc. ACM HPCVirt* (2009), pp. 17–24.

[9] JONES, M., ROSU, D., AND ROSU, M.-C. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. In *Proc. ACM SOSP* (1997), pp. 198–211.

[10] KRASIC, C., SAUBHASIK, M., AND GOEL, A. Fair and Timely Scheduling via Cooperative Polling. In *Proc. ACM EuroSys* (2009), pp. 103–116.

[11] LAGAR-CAVILLA, H., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. VMM-Independent Graphics Acceleration. In *Proc. ACM VEE* (2007), pp. 33–43.

[12] LEHOCZKY, J., SHA, L., AND STROSNIDER, J. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proc. IEEE RTSS* (1987), pp. 261–270.

[13] LIU, C., AND LAYLAND, J. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM 20* (1973), 46–61.

[14] MARTIN, K., FAITH, R., OWEN, J., AND AKIN, A. *Direct Rendering Infrastructure, Low-Level Design Document*. Precision Insight, Inc., 1999.

[15] MESA3D. Gallium3D. `http://www.mesa3d.org/`.

[16] MOLANO, A., JUWA, K., AND RAJKUMAR, R. Real-Time Filesystems. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. In *Proc. IEEE RTSS* (1997), pp. 155–165.

[17] NIEH, J., AND LAM, M. SMART: A Processor Scheduler for Multimedia Applications. In *Proc. ACM SOSP* (1995).

[18] NIGHTINGALE, E., HODSON, O., MCLLORY, R., HAWBLITZEL, C., AND HUNT, G. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proc. ACM SOSP* (2009).

[19] NVIDIA CORPORATION. Proprietary Driver. `http://www.nvidia.com/page/drivers.html`.

[20] OIKAWA, S., AND RAJKUMAR, R. Portable RT: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. In *Proc. IEEE RTAS* (1999), pp. 111–120.

[21] PATHSCALE INC. ENZO. `http://www.pathscale.com/`.

[22] PATHSCALE INC. PSCNV. `https://github.com/pathscale/pscnv`.

[23] PHORONIX. NVIDIA Developer Talks Openly About Linux Support. `http://www.phoronix.com/scan.php?page=article&item=nvidia_ga_linux&num=2`.

[24] PHORONIX. Phoronix Test Suite. `http://www.phoronix-test-suite.com/`.

[25] PRONOVOST, S., MORETON, H., AND KELLEY, T. Windows Display Driver Model (WDDM v2) And Beyond. In *Windows Hardware Engineering Conference* (2006).

[26] RAJKUMAR, R., LEE, C., LEHOCZKY, J., AND SIEWIOREK, D. A Resource Allocation Model for QoS Management. In *Proc. IEEE RTSS* (1997), pp. 298–307.

[27] ROUSSOS, K., BITAR, N., AND ENGLISH, R. Deterministic Batch Scheduling Without Static Partitioning. In *Proc. JSSPP* (1999), pp. 220–235.

[28] SPEC. SPECviewperf. `http://www.spec.org/gwpg/gpc.static/vp11info.html`.

[29] SPRUNT, B., LEHOCZKY, J., AND SHA, L. Exploiting Unused Periodic Time for Aperiodic Service using the Extended Priority Exchange Algorithm. In *Proc. IEEE RTSS* (1988), pp. 251–258.

[30] SPURI, M., AND BUTTAZO, G. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proc. IEEE RTSS* (1994), pp. 2–11.

[31] YANG, T., LIU, T., BERGER, E., KAPLAN, S., AND MOSS, J.-B. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *Proc. USENIX OSDI* (2008), pp. 73–86.

Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems

Vishakha Gupta
Georgia Institute of Technology

Karsten Schwan
Georgia Institute of Technology

Niraj Tolia
Maginatics

Vanish Talwar
HP Labs

Parthasarathy Ranganathan
HP Labs

Abstract

Heterogeneous multi-cores—platforms comprised of both general purpose and accelerator cores—are becoming increasingly common. While applications wish to freely utilize all cores present on such platforms, operating systems continue to view accelerators as specialized devices. The Pegasus system described in this paper uses an alternative approach that offers a uniform resource usage model for all cores on heterogeneous chip multiprocessors. Operating at the hypervisor level, its novel scheduling methods fairly and efficiently share accelerators across multiple virtual machines, thereby making accelerators into first class schedulable entities of choice for many-core applications. Using NVIDIA GPGPUs coupled with x86-based general purpose host cores, a Xen-based implementation of Pegasus demonstrates improved performance for applications by better managing combined platform resources. With moderate virtualization penalties, performance improvements range from 18% to 140% over base GPU driver scheduling when the GPUs are shared.

1 Introduction

Systems with specialized processors like those used for accelerating computations, network processing, or cryptographic tasks [27, 34] have proven their utility in terms of higher performance and lower power consumption. This is not only causing tremendous growth in accelerator-based platforms, but it is also leading to the release of heterogeneous processors where x86-based cores and on-chip network or graphics accelerators [17, 31] form a common pool of resources. However, operating systems and virtualization platforms have not yet adjusted to these architectural trends. In particular, they continue to treat accelerators as secondary devices and focus scheduling and resource management on their general purpose processors, supported by vendors that shield developers from the complexities of accelerator hardware by 'hiding' it behind drivers that only expose higher level programming APIs [19, 28]. Unfortunately, technically, this implies that drivers rather than operating systems or hypervisors determine how accelerators are shared, which restricts scheduling policies and thus, the optimization criteria applied when using such heterogeneous systems.

A driver-based execution model can not only potentially hurt utilization, but also make it difficult for applications and systems to obtain desired benefits from the combined use of heterogeneous processing units. Consider, for instance, an advanced image processing service akin to HP's Snapfish [32] or Microsoft's PhotoSynth [25] applications, but offering additional computational services like complex image enhancement and watermarking, hosted in a data center. For such applications, the low latency responses desired by end users require the combined processing power of both general purpose and accelerator cores. An example is the execution of sequences of operations like those that first identify spatial correlation or correspondence [33] between images prior to synthesizing them [25]. For these pipelined sets of tasks, some can efficiently run on multicore CPUs, whereas others can substantially benefit from acceleration [6, 23]. However, when they concurrently use both types of processing resources, low latency is attained only when different pipeline elements are appropriately co- or gang-scheduled onto both CPU and GPU cores. As shown later in this paper, such co-scheduling is difficult to perform with current accelerators when used in consolidated data center settings. Further, it is hard to enforce fairness in accelerator use when the many clients in typical web applications cause multiple tasks to compete for both general purpose and accelerator resources,

The *Pegasus* project addresses the urgent need for systems support to smartly manage accelerators. It does this by leveraging the new opportunities presented by increased adoption of virtualization technology in commercial, cloud computing [1], and even high performance infrastructures [22, 35]: *the Pegasus hypervisor*

extensions (1) make accelerators into first class schedulable entities and (2) support scheduling methods that enable efficient use of both the general purpose and accelerator cores of heterogeneous hardware platforms. Specifically, for platforms comprised of x86 CPUs connected to NVIDIA GPUs, these extensions can be used to manage all of the platform's processing resources, to address the broad range of needs of GPGPU (general purpose computation on graphics processing units) applications, including the high throughput requirements of compute intensive web applications like the image processing code outlined above and the low latency requirements of computational finance [24] or similarly computationally intensive high performance codes. For high throughput, platform resources can be shared across many applications and/or clients. For low latency, resource management with such sharing also considers individual application requirements, including those of the inter-dependent pipeline-based codes employed for the financial and image processing applications.

The Pegasus hypervisor extensions described in Sections 3 and 5 do not give applications direct access to accelerators [28], nor do they hide them behind a virtual file system layer [5, 15]. Instead, similar to past work on self-virtualizing devices [29], Pegasus exposes to applications a virtual accelerator interface, and it supports existing GPGPU applications by making this interface identical to NVIDIA's CUDA programming API [13]. As a result, whenever a virtual machine attempts to use the accelerator by calling this API, control reverts to the hypervisor. This means, of course, that the hypervisor 'sees' the application's accelerator accesses, thereby getting an opportunity to regulate (schedule) them. A second step taken by Pegasus is to then explicitly *coordinate* how VMs use general purpose and accelerator resources. With the Xen implementation [7] of Pegasus shown in this paper, this is done by explicitly scheduling guest VMs' accelerator accesses in Xen's Dom0, while at the same time controlling those VMs' use of general purpose processors, the latter exploiting Dom0's privileged access to the Xen hypervisor and its VM scheduler.

Pegasus elevates accelerators to first class schedulable citizens in a manner somewhat similar to the way it is done in the Helios operating system [26], which uses satellite kernels with standard interfaces for XScale-based IO cards. However, given the fast rate of technology development in accelerator chips, we consider it premature to impose a common abstraction across all possible heterogeneous processors. Instead, Pegasus uses a more loosely coupled approach in which it assumes systems to have different 'scheduling domains', each of which is adept at controlling its own set of resources, e.g., accelerator vs. general purpose cores. Pegasus scheduling, then, coordinates when and to what extent, VMs use the resources managed by these multiple scheduling domains. This approach leverages notions of 'cellular' hypervisor structures [11] or federated schedulers that have been shown useful in other contexts [20]. Concurrent use of both CPU and GPU resources is one class of coordination methods Pegasus implements, with other methods aimed at delivering both high performance and fairness in terms of VM usage of platform resources.

Pegasus relies on application developers or toolchains to identify the right target processors for different computational tasks and to generate such tasks with the appropriate instruction set architectures (ISAs). Further, its current implementation does not interact with tool chains or runtimes, but we recognize that such interactions could improve the effectiveness of its runtime methods for resource management [8]. An advantage derived from this lack of interaction, however, is that Pegasus does not depend on certain toolchains or runtimes, nor does it require internal information about accelerators [23]. As a result, Pegasus can operate with both 'closed' accelerators like NVIDIA GPUs and with 'open' ones like IBM Cell [14], and its approach can easily be extended to support other APIs like OpenCL [19].

Summarizing, the Pegasus hypervisor extensions make the following contributions:

Accelerators as first class schedulable entities—accelerators (accelerator physical CPUs or aPCPUs) can be managed as first class schedulable entities, i.e., they can be shared by multiple tasks, and task mappings to processors are dynamic, within the constraints imposed by the accelerator software stacks.

Visible heterogeneity—Pegasus respects the fact that aPCPUs differ in capabilities, have different modes of access, and sometimes use different ISAs. Rather than hiding these facts, Pegasus exposes heterogeneity to the applications and the guest virtual machines (VMs) that are capable of exploiting it.

Diversity in scheduling—accelerators are used in multiple ways, e.g., to speedup parallel codes, to increase throughput, or to improve a platform's power/performance properties. Pegasus addresses differing application needs by offering a diversity of methods for scheduling accelerator and general purpose resources, including co-scheduling for concurrency constraints.

'Coordination' as the basis for resource management—internally, accelerators use specialized execution environments with their own resource managers [14, 27]. Pegasus uses *coordinated scheduling methods* to align accelerator resource usage with platform-level management. While coordination applies external controls to control the use of 'closed' accelerators, i.e., accelerators with resource managers that do not export coordination interfaces, it could interact more intimately with 'open' managers as per their internal scheduling methods.

Novel scheduling methods—current schedulers on parallel machines assume complete control over their underlying platforms' processing resources. In contrast, Pegasus recognizes and deals with heterogeneity not only in terms of differing resource capabilities, but also in terms of the diverse scheduling methods these resources may require, an example being the highly parallel internal scheduling used in GPGPUs. Pegasus coordination methods, therefore, differ from traditional co-scheduling in that they operate above underlying native techniques. Such meta-scheduling, therefore, seeks to influence the actions of underlying schedulers rather than replacing their functionality. This paper proposes and evaluates new coordination methods that are geared to dealing with diverse resources, including CPUs vs. GPUs and multiple generations of the latter, yet at the same time, attempting to preserve desired virtual platform properties, including fair-sharing and rioritization.

The current Xen-based Pegasus prototype efficiently virtualizes NVIDIA GPUs, resulting in performance competitive with that of applications that have direct access to the GPU resources, as shown in Section 6. More importantly, when the GPGPU resources are shared by multiple guest VMs, online resource management becomes critical. This is evident from the performance benefits derived from the coordination policies described in Section 4, which range from 18% to 140% over base GPU driver scheduling. An extension to the current, fully functional, single-node Pegasus prototype will be deployed to a large-scale GPU-based cluster machine, called Keeneland, under construction at Oak Ridge National Labs [35], to further validate our approach and to better understand how to improve the federated scheduling infrastructures needed for future larger scale heterogeneous systems.

In the remaining paper, Section 2 articulates the need for smart accelerator sharing. Section 3 outlines the Pegasus architecture. Section 4 describes its rich resource management methods. A discussion of scheduling policies is followed by implementation details in Section 5, and experimental evaluation in Section 6. Related work is in Section 7, followed by conclusions and future work.

2 Background

This section offers additional motivation for the Pegasus approach on a heterogeneous multi-core platforms.

Value in sharing resources—Accelerator performance and usability (e.g., the increasing adoption of CUDA) are improving rapidly. However, even for today's platforms, the majority of applications do not occupy the entire accelerator [2, 18]. In consequence and despite continuing efforts to improve the performance of single accelerator applications [12], resource sharing is now supported in NVIDIA's Fermi architecture [27], IBM's Cell, and others. These facts are the prime drivers behind our decision to develop scheduling methods that can efficiently utilize both accelerator and general purpose cores. However, as stated earlier, for reasons of portability across different accelerators and accelerator generations, and to deal with their proprietary nature, Pegasus resource sharing across different VMs is implemented at a layer above the driver, leaving it up to the individual applications running in each VM to control and optimize their use of accelerator resources.

Limitations of traditional device driver based solutions—Typical accelerators have a sophisticated and often proprietary device driver layer, with an optional runtime. While these efficiently implement the computational and data interactions between accelerator and host cores [28], they lack support for efficient resource sharing. For example, first-come-first-serve issue of CUDA calls from 'applications-to-GPU' through a centralized NVIDIA-driver can lead to possibly detrimental call interleavings, which can cause high variances in call times and degradation in performance, as shown by measurements of the NVIDIA driver in Section 6. Pegasus can avoid such issues and use a more favorable call order, by introducing and regulating time-shares for VMs to issue GPU-requests. This leads to significantly improved performance even for simple scheduling schemes.

3 Pegasus System Architecture

Designed to generalize from current accelerator-based systems to future heterogeneous many-core platforms, Pegasus creates the logical view of computational resources shown in Figure 1. In this view, general purpose and accelerator tasks are schedulable entities mapped to VCPUs (virtual CPUs) characterized as general purpose or as 'accelerator'. Since both sets of processors can be scheduled independently, platform-wide scheduling, then, requires Pegasus to federate the platform's general purpose and accelerator schedulers. Federation is implemented by coordination methods that provide the serviced virtual machines with shares of physical processors based on the diverse policies described in Section 4. *Coordination is particularly important for closely coupled tasks running on both accelerator and general purpose cores*, as with the image processing application explained earlier. Figure 1 shows virtual machines running on either one or both types of processors, i.e., the CPUs and/or the accelerators. The figure also suggests the relative rarity of VMs running solely on accelerators (grayed out in the figure) in current systems. We segregate the privileged software components shown for the host and accelerator cores to acknowledge that the accelerator could have its own privileged runtime.

The following questions articulate the challenges in achieving the vision shown in Figure 1.

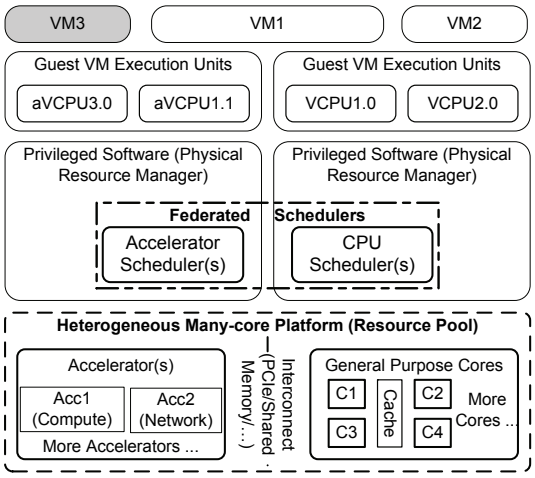


Figure 1: Logical view of Pegasus architecture

How can heterogeneous resources be managed?: Hardware heterogeneity goes beyond varying compute speeds to include differing interconnect distances, different and possibly disjoint memory models, and potentially different or non-overlapping ISAs. This makes it difficult to assimilate these accelerators into one common platform. Exacerbating these hardware differences are software challenges, like those caused by the fact that there is no general agreement about programming models and runtimes for accelerator-based systems [19, 28].

Are there efficient methods to utilize heterogeneous resources?: The hypervisor has limited control over how the resources internal to closed accelerators are used, and whether sharing is possible in time, space, or both because there is no direct control over scheduler actions beyond the proprietary interfaces. The concrete question, then, is whether and to what extent the coordinated scheduling approach adopted by Pegasus can succeed.

Pegasus therefore allows schedulers to run resource allocation policies that offer diversity in how they maximize application performance and/or fairness in resource sharing.

3.1 Accelerator Virtualization

With GViM [13], we outline methods for low-overhead virtualization of GPUs for the Xen hypervisor, addressing heterogeneous hardware with general purpose and accelerator cores, used by VMs with suitable codes (e.g., for Larrabee or Tolapai cores, codes that are IA instruction set compatible vs. non-IA compatible codes for NVIDIA or Cell accelerators). Building on this approach and acknowledging the current off-chip nature of accelerators, Pegasus assumes these hardware resources to be managed by both the hypervisor and Xen's 'Dom0' management (and driver) domain. Hence, Pegasus uses front end/back end split drivers [3] to mediate all accesses to GPUs connected via PCIe. Specifically, the requests for GPU usage issued by guest VMs (i.e., CUDA tasks) are contained in call buffers shared between guests and Dom0, as shown in Figure 2, using a separate buffer for each guest. Buffers are inspected by 'poller' threads that pick call packets from per-guest buffers and issue them to the actual CUDA runtime/driver resident in Dom0. These poller threads can be woken up whenever a domain has call requests waiting. This model of execution is well-matched with the ways in which guests use accelerators, typically wishing to utilize their computational capabilities for some time and with multiple calls.

For general purpose cores, a VCPU as the (virtual) CPU representation offered to a VM embodies the state representing the execution of the VM's threads/processes on physical CPUs (PCPUs). As a similar abstraction, Pegasus introduces the notion of an *accelerator VCPU (aVCPU)*, which embodies the VM's state concerning the execution of its calls to the accelerator. For the Xen/NVIDIA implementation, this abstraction is a combination of state allocated on the host and on the accelerator (i.e., Dom0 polling thread, CUDA calls, and driver context form the execution context while the data that is operated upon forms the data portion, when compared with the VCPUs). By introducing aVCPUs, Pegasus can then explicitly schedule them, just like their general purpose counterparts. Further, and as seen from Section 6, virtualization costs are negligible or low and with this API-based approach to virtualization, Pegasus leaves the use of resources on the accelerator hardware up to the application, ensures portability and independence from low-level changes in NVIDIA drivers and hardware.

3.2 Resource Management Framework

For VMs using both VCPUs and aVCPUs, resource management can explicitly track and schedule their joint use of both general purpose and accelerator resources. Technically, such management involves scheduling their VCPUs and aVCPUs to meet desired Service Level Objectives (SLOs), concurrency constraints, and to ensure fairness in different guest VMs' resource usage.

For high performance, Pegasus distinguishes two phases in accelerator request scheduling. First, the **accelerator selection module** runs in the Accelerator Domain—which in our current implementation is Dom0—henceforth, called *DomA*. This module associates a domain, i.e., a guest VM, with an accelerator that has available resources, by placing the domain into an 'accelerator ready queue', as shown in Figure 2. Domains are selected from this queue when they are ready to issue requests. Second, it is only after this selection that actual usage requests are forwarded to, i.e., scheduled and run on, the selected accelerator. There are multiple reasons for this difference in accelerator vs. CPU scheduling. (1) An accelerator like the NVIDIA GPU

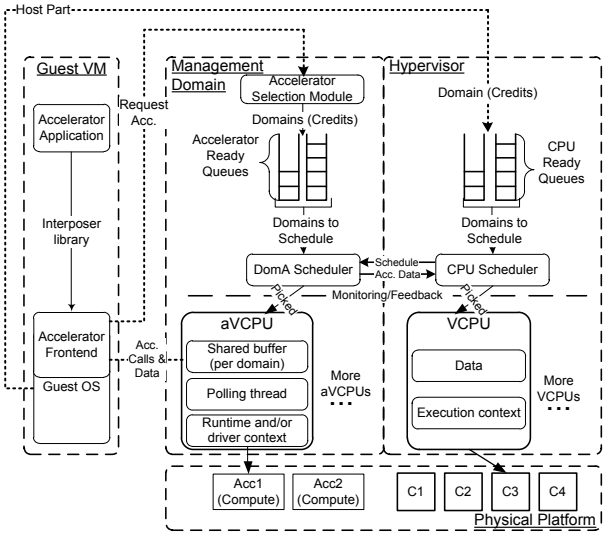


Figure 2: Logical view of the resource management framework in Pegasus

has limited memory, and it associates a context with each 'user' (e.g., a thread) that locks some of the GPU's resources. (2) Memory swapping between host and accelerator memory over an interconnect like PCIe is expensive, which means that it is costly to dynamically change the context currently running on the GPU. In response, Pegasus GPU scheduling restricts the number of domains simultaneously scheduled on each accelerator and in addition, it permits each such domain to use the accelerator for some extensive time duration. The following parameters are used for accelerator selection.

Accelerator profile and queue—accelerators vary in terms of clock speed, memory size, in-out bandwidths and other such physical characteristics. These are static or hardware properties that can identify capability differences between various accelerators connected in the system. There also are dynamic properties like allocated memory, number of associated domains, etc., at any given time. This static and dynamic information is captured in an 'accelerator profile'. An 'accelerator weight' computed from this profile information determines current hardware capabilities and load characteristics for the accelerator. These weights are used to order accelerators in a priority queue maintained within the DomA Scheduler, termed as 'accelerator queue'. For example, the more an accelerator is used, the lower its weight becomes so that it does not get oversubscribed. The accelerator with the highest weight is the most capable and is the first to be considered when a domain requests accelerator use.

Domain profile—domains may be more or less demanding of accelerator resources and more vs. less capable of using them. The 'domain profiles' maintained by Pegasus describe these differences, and they also quantitatively capture domain requirements. Concretely, the current implementation expects credit assignments [7] for each domain that gives it proportional access to the accelerator. Another example is to match the domain's expected memory requirements against the available memory on an accelerator (with CUDA, it is possible to determine this from application metadata). Since the execution properties of domains change over time, domain execution characteristics should be determined dynamically, which would then cause the runtime modification of a domain's accelerator credits and/or access privileges to accelerators. Automated methods for doing so, based on runtime monitoring, are subject of our future work, with initial ideas reported in [8]. This paper lays the groundwork for such research: (1) we show coordination to be a fundamentally useful method for managing future heterogeneous systems, and (2) we demonstrate the importance of these runtime-based techniques and performance advantages derived from their use in a coordinated scheduling environment.

Once a domain has been associated with an accelerator, the **DomA Scheduler** in Figure 2 schedules execution of individual domain requests per accelerator by activating the corresponding domain's aVCPU. For all domains in its ready queue, the 'DomA Scheduler' has complete control over which domain's requests are submitted to the accelerator(s), and it can make such decisions in coordination with the hypervisor's VCPU scheduler, by exchanging relevant accelerator and schedule data. Scheduling in this second phase, can thus be enhanced by *coordinating* the actions of the hypervisor and DomA scheduler(s) present on the platform, as introduced in Figure 1. In addition, certain coordination policies can use the monitoring/feedback module, which currently tracks the average values of wait times for accelerator requests, the goal being to detect SLO (service level objective) violations for guest requests. Various policies supported by the DomA scheduler are described in the following section.

4 Resource Management Policies for Heterogeneity-aware Hypervisors

Pegasus contributes its novel, federated, and heterogeneity-aware scheduling methods to the substantive body of past work in resource management. The policies described below, and implemented by the DomA scheduler, are categorized based on their level of interaction with the hypervisor's scheduler. They range from simple and easily implemented schemes offering basic scheduling properties to coordination-based policies that exploit information sharing between the hypervisor and accelerator subsystems. *Policies are designed to demonstrate the range of achievable coordination between the two scheduler subsystems*

Algorithm 1: Simplified Representation of Scheduling Data and Functions for Credit-based Schemes

```
/* D = Domain being considered */
/* X = Domain cpu or accelerator credits */
/* T = Scheduler timer period */
/* Tc = Ticks assigned to next D */
/* Tm = maximum ticks D gets based on X */
Data: Ready queue RQ_A of domains (D)
/* RQ is ordered by X */
Data: Accelerator queue AccQ of accelerators
/* AccQ is ordered by accelerator weight */

InsertDomainforScheduling(D)
  if D not in RQ_A then
      T_c ← 1, T_m ← X / X_min
      A ← PickAccelerator(AccQ,D)
      InsertDomainInRQ_CreditSorted(RQ_A,D)
  else
      /* D already in some RQ_A */
      if ContextEstablished then
          T_c ← T_m
      else
          T_c ← 1

DomASchedule(RQ_A)
  InsertDomainforScheduling(Curr_Dom)
  D ← RemoveHeadandAdvance(RQ_A)
  Set D's timer period to T_c; Curr_dom ← D
```

Algorithm 2: Simplified Representation of CoSched and AugC Schemes

```
/* RQ_cpu=Per CPU ready q in hypervisor */
/* HS=VCPU-PCPU schedule for next period */
/* X = domain credits */

HypeSchedule(RQ_cpu)
  Pick VCPUs for all PCPUs in system
  ∀D, AugCredit_D = RemainingCredit
  Pass HS to DomA scheduler

DomACoSchedule(RQ_A, HS)
  /* To handle #cpus > #accelerators */
  ∀D ∈ (RQ_A ⋂ HS)
    Pick D with highest X
  if D = null then
      /* To improve GPU utilization */
      Pick D with highest X in RQ_A

DomAAugSchedule(RQ_A, HS)
  foreach D ∈ RQ_A do
    Pick D with highest (AugCredit + X)
```

and the benefits seen by such coordination for various workloads. The specific property offered by each policy is indicated in square brackets.

4.1 Hypervisor Independent Policies

The simplest methods do not support scheduler federation, limiting their scheduling logic to DomA.

No scheduling in backend (None) [first-come-first-serve]—provides base functionality that assigns domains to accelerators in a round robin manner, but relies on NVIDIA's runtime/driver layer to handle all request scheduling. DomA scheduler plays no role in domain request scheduling. This serves as our baseline.

AccCredit (AccC) [proportional fair-share]—recognizing that domains differ in terms of their desire and ability to use accelerators, accelerator credits are associated with each domain, based on which different domains are polled for different time periods. This makes the time given to a guest proportional to how much it desires to use the accelerator, as apparent in the pseudo-code shown in Algorithm 1, where the requests from the domain at the head of the queue are handled until it finishes its awarded number of ticks. For instance, with credit assignments (Dom1,1024), (Dom2,512), (Dom3,256), and (Dom4,512), the number of ticks will be 4, 2, 1, and 2, respectively.

Because the accelerators used with Pegasus require their applications to explicitly allocate and free accelerator state, it is easy to determine whether or not a domain currently has context (state) established on an accelerator. The DomA scheduler, therefore, interprets a domain in a ContextEstablished state as one that is actively using the accelerator. When in a NoContextEstablished state, a minimum time tick (1) is assigned to the domain for the next scheduling cycle (see Algorithm 1).

4.2 Hypervisor Controlled Policy

The rationale behind coordinating VCPUs and aVCPUs is that the overall execution time of an application (comprised of both host and accelerator portions) can be reduced if its communicating host and accelerator tasks are scheduled at the same time. We implement one such method described next.

Strict co-scheduling (CoSched) [latency reduction by occasional unfairness]—an alternative to the accelerator-centric policies shown above, this policy gives complete control over scheduling to the hypervisor. Here, accelerator cores are treated as slaves to host cores, so that VCPUs and aVCPUs are scheduled at the same time. This policy works particularly well for latency-sensitive workloads like certain financial processing codes [24] or barrier-rich parallel applications. It is implemented by permitting the hypervisor scheduler to control how DomA schedules aVCPUs, as shown in Algorithm 2. For 'singular VCPUs', i.e., those without associated aVCPUs, scheduling reverts to using a standard credit-based scheme.

4.3 Hypervisor Coordinated Policies

A known issue with co-scheduling is potential unfairness. The following methods have the hypervisor actively participate in making scheduling decisions rather than governing them:

Augmented credit-based scheme (AugC) [throughput improvement by temporary credit boost]—going

beyond the proportionality approach in AccC, this policy uses active coordination between the DomA scheduler and hypervisor (Xen) scheduler in an attempt to better co-schedule domains on a CPU and GPU. To enable coscheduling, the Xen credit-based scheduler provides to the DomA scheduler, as a hint, its CPU schedule for the upcoming period, with remaining credits for all domains in the schedule as shown in Algorithm 2. The DomA scheduler uses this schedule to add temporary credits to the corresponding domains in its list (i.e., to those that have been scheduled for the next CPU time period). This boosts the credits of those domains that have their VCPUs selected by CPU scheduling, thus increasing their chances for getting scheduled on the corresponding GPU. While this effectively co-schedules these domains' CPU and GPU tasks, the DomA scheduler retains complete control over its actions; no domain with high accelerator credits is denied its eventual turn due to this temporary boost.

SLA feedback to meet QoS requirements (SLAF) [feedback-based proportional fair-share]—this is an adaptation of the AccC scheme as shown in Algorithm 1, with feedback control. (1) We start with an SLO defined for a domain (statically profiled) as the expected accelerator utilization—e.g., 0.5sec every second. (2) As shown in Algorithm 1, once the domain moves to a ContextEstablished state, it is polled, and its requests are handled for its assigned duration. In addition, a sum of domain poll time is maintained. (3) Ever so often, all domains associated with an accelerator are scanned for possible SLO violations. Domains with violations are given extra time ticks to compensate, one per scheduling cycle. (4) In high load conditions, there is a trigger that increases accelerator load in order to avoid new domain requests, which in the worst case, forces domains with comparatively low credits to wait longer to get compensated for violations seen by higher credit domains.

For generality in scheduling, we have also implemented: (1) Round robin (RR) [fair-share] which is hypervisor independent, and (2) XenoCredit (XC) [proportional fair-share] which is similar to AccC except it depends on CPU credits assigned to the corresponding VM, making it a hypervisor coordinated policy.

5 System Implementation

The current Pegasus implementation operates with Xen and NVIDIA GPUs. As a result, resource management policies are implemented within the management framework (Section 3.2) run in DomA (i.e., Dom0 in the current implementation), as shown in Figure 2.

Discovering GPUs and guest domains: the management framework discovers all of the GPUs present in the system, assembles their static profiles using *cudaGetDeviceProperties()* [28], and registers them with the Pegasus hypervisor scheduling extensions. When new guest domains are created, Xen adds them to its hypervisor scheduling queue. Our management framework, in turn, discovers them by monitoring XenStore.

Scheduling: the scheduling policies RR, AccC, XC, and SLAF are implemented using timer signals, with one tick interval equal to the hypervisor's CPU scheduling timer interval. There is one timer handler or scheduler for each GPU, just like there is one scheduling timer interrupt per CPU, and this function picks the next domain to run from corresponding GPU's ready queue, as shown in Algorithm 1. AugC and CoSched use a thread in the backend that performs scheduling for each GPU by checking the latest schedule information provided by the hypervisor, as described in Section 4. It then sleeps for one timer interval. The per domain pollers are woken up or put to sleep by scheduling function(s), using real time signals with unique values assigned to each domain. This restricts the maximum number of domains supported by the backend to the Dom0 operating system imposed limit, but results in bounded/prioritized signal delivery times.

Two entirely different scheduling domains, i.e., DomA and the hypervisor, control the two different kinds of processing units, i.e., GPUs and x86 cores. This poses several implementation challenges for the AugC and CoSched policies such as: (1) What data needs to be shared between extensions and the hypervisor scheduler and what additional actions to take, if any, in the hypervisor scheduler, given that this scheduler is in the critical path for the entire system? (2) How do we manage the differences and drifts in these respective schedulers' time periods?

Concerning (1), the current implementation extends the hypervisor scheduler to simply have it share its VCPU-PCPU schedule with the DomA scheduler, which then uses this schedule to find the right VM candidates for scheduling. Concerning (2), there can be a noticeable timing gap between when decisions are made and then enacted by the hypervisor scheduler vs. the DomA extensions. The resulting delay as to when or how soon a VCPU and an aVCPU from same domain are co-scheduled can be reduced with better control over the use of GPU resources. Since NVIDIA drivers do not offer such control, there is notable variation in co-scheduling. Our current remedial solution is to have each aVCPU be executed for 'some time', i.e., to run multiple CUDA call requests, rather than scheduling aVCPUs at a per CUDA call granularity, thereby increasing the possible overlap time with its 'co-related' VCPU. This does not solve the problem, but it mitigates the effects of imprecision, particularly for longer running workloads.

6 Experimental Evaluation

Key contributions of Pegasus are (1) accelerators as first class schedulable entities and (2) coordinated scheduling to provide applications with the high levels of performance sought by use of heterogeneous processing resources. This section first shows that the Pegasus way of virtualizing accelerators is efficient, next demonstrates the importance of coordinated resource management, and finally, presents a number of interesting insights about how diverse coordination (i.e., scheduling) policies can be used to address workload diversity.

Testbed: All experimental evaluations are conducted on a system comprised of (1) a 2.5GHz Xeon quad-core processor with 3GB memory and (2) an NVIDIA 9800 GTX card with 2 GPUs and the v169.09 GPU driver. The Xen 3.2.1 hypervisor and the 2.6.18 Linux kernel are used in Dom0 and guest domains. Guest domains use 512MB memory and 1 VCPU each, the latter pinned to certain physical cores, depending on the experiments being conducted.

6.1 Benchmarks and Applications

Pegasus is evaluated with an extensive set of benchmarks and with emulations of more complex computationally expensive enterprise codes like the web-based image processing application mentioned earlier. Benchmarks include (1) parallel codes requiring low levels of deviation for highly synchronous execution, and (2) throughput-intensive codes. A complete listing appears in Table 1, identifying them as belonging to either the parboil benchmark suite [30] or the CUDA SDK 1.1. Benchmark-based performance studies go beyond running individual codes to using representative code mixes that have varying needs and differences in behavior due to different dataset sizes, data transfer times, iteration complexity, and numbers of iterations executed for certain computations. The latter two are a good measure of GPU 'kernel' size and the degree of coupling between CPUs orchestrating accelerator use and the GPUs running these kernels respectively. Depending on their outputs and the number of CUDA calls made, (1) throughput-sensitive benchmarks are MC, BOp, PI, (2) latency-sensitive benchmarks include FWT, and scientific, and (3) some benchmarks are both, e.g., BS, CP. A benchmark is throughput-sensitive when its performance is best evaluated as the number of some quantity processed or calculated per second, and a benchmark is latency-sensitive when it makes frequent CUDA calls and its execution time is sensitive to potential virtualization overhead and/or delays or 'noise' in accelerator scheduling. The image processing application, termed PicSer, emulates web codes like PhotoSynth. BlackScholes represents financial codes like those run by option trading companies [24].

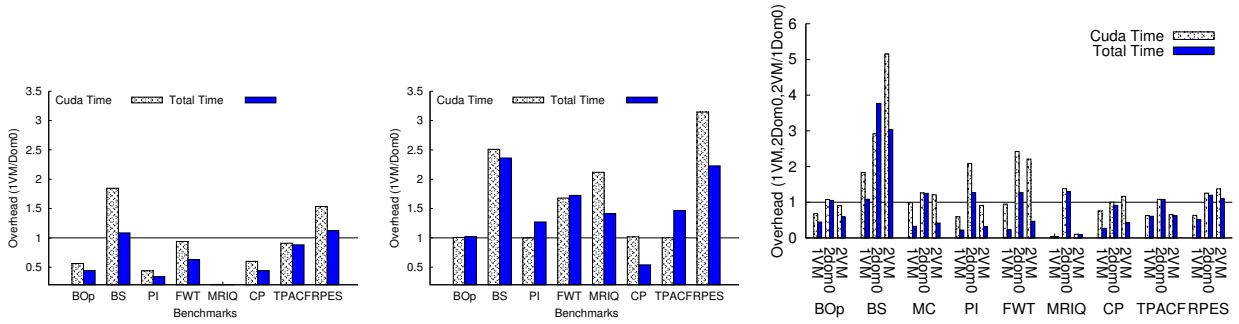
Category	Source	Benchmarks
Financial	SDK	Binomial(BOp), BlackScholes(BS), MonteCarlo(MC)
Media processing	SDK or parboil	ProcessImage(PI)=matrix multiply+DXTC, MRIQ, FastWalshTransform(FWT)
Scientific	parboil	CP, TPACF, RPES

Table 1: Summary of Benchmarks

6.2 GPGPU Virtualization

Virtualization overheads when using Pegasus are depicted in Figures 3(a)–(c), using the benchmarks listed in Table 1. Results show the overhead (or speedup) when running the benchmark in question in a VM vs. when running it in Dom0. The overhead is calculated as the time it takes the benchmark to run in a VM divided by the time to run it in Dom0. We show the overhead (or speedup) for the average total execution time (Total Time) and the average time for CUDA calls (Cuda Time) across 50 runs of each benchmark. Cuda Time is calculated as the time to execute all CUDA calls within the application. Running the benchmark in Dom0 is equivalent to running it in a non-virtualized setting. For the 1VM numbers in Figure 3(a) and (c), all four cores are enabled, and to avoid scheduler interactions, Dom0 and the VM are pinned on separate cores. The experiments reported in Figure 3(b) have only 1 core enabled and the execution times are not averaged over multiple runs, with a backend restart for every run. This is done for reasons explained next. All cases use an equal number of physical GPUs, and Dom0 tests are run with as many cores as the Dom0–1VM case.

An interesting observation about these results is that sometimes, it is better to use virtualized rather than non-virtualized accelerators. This is because (1) the Pegasus virtualization software can benefit from the concurrency seen from using different cores for the guest vs. Dom0 domains, and (2) further advantages are derived from additional caching of data due to a constantly running—in Dom0—backend process and NVIDIA driver. This is confirmed in Figure 3(b), which shows higher overheads when the backend is stopped before every run, wiping out any driver cache information. Also of interest is the speedup seen by say, BOp or PI vs. the performance seen by say, BS or RPES, in Figure 3(a). This is due to an increase in the number of calls per application, seen in BOp/PI vs. BS/RPES, emphasizing the virtualization overhead added to each executed CUDA call. In these cases, the benefits from caching and the presence of multiple cores are outweighed by the per call overhead multiplied by the number of calls made.



a) On a quad-core b) On 1 core - no driver caching c) With GPU sharing (scheduler - RR)

Figure 3: Evaluation of GPU virtualization overhead (lower is better)

6.3 Resource Management

The Pegasus framework for scheduling coordination makes it possible to implement diverse policies to meet different application needs. Consequently, we use multiple metrics to evaluate the policies described in Section 4. They include (1) throughput (*Quantity/sec*) for throughput-sensitive applications, (2) work done (*Quantity work/sec*) (which is the sum of the calculations done over all runs divided by the total time taken), and/or (3) per call latency (Latency) observed for CUDA calls (latency is reported including the CUDA function execution time to account for the fact that we cannot control how the driver orders the requests it receives from Pegasus).

Experimental Methodology: To reduce scheduling interference from the guest OS, each VM runs only a single benchmark. Each sample set of measurements, then, involves launching the required number of VMs, each of which repeatedly runs its benchmark. To evaluate accelerator sharing, experiments use 2, 3, or 4 domains, which translates to configurations with no GPU/CPU sharing, sharing of one GPU and one CPU by two of the three domains, and sharing of two CPUs and both GPUs by pairs of domains, respectively. In all experiments, Dom1 and Dom3 share a CPU as well as a GPU, and so do Dom2 and Dom4, when present. Further, to avoid non-deterministic behavior due to actions taken by the hypervisor scheduler, and to deal with the limited numbers of cores and GPGPUs available on our experimental platform, we pin the domain VCPUs to certain CPUs, depending on the experiment scenario. These CPUs are chosen based on the workload distribution across CPUs (including backend threads in Dom0) and the concurrency requirements of VCPU and aVCPU from the same domain (simply put, VCPU from a domain and the polling thread forming its aVCPU cannot be co-scheduled if they are bound to the same CPU).

For brevity, the results shown in this section focus on the BS benchmark, because of (1) its closeness to real world financial workloads, (2) its tunable iteration count argument that varies its CPU-GPU coupling and can highlight the benefits of coordination, (3) its easily varied data sizes and hence GPU computation complexity, and (4) its throughput as well as latency sensitive nature. Additional reported results are for benchmarks like PicSer, CP and FWT in order to highlight specific interesting/different cases, like those for applications with low degrees of coupling or with high latency sensitivity. For experiments that assign equal credits to all domains, we do not plot RR and AccC, since they are equivalent to XC. Also, we do not show AccC if accelerator credits are equal to Xen credits.

Observations at an early stage of experimentation showed that the CUDA driver introduces substantial variations in execution time when a GPU is shared by multiple applications (shown by the NoVirt graph in Figure 9). This caused us to use a large sample size of 50 runs per benchmark per domain, and we report either the h-spread[1] or work done which is the sum of total output divided by total elapsed time over those multiple runs. For throughput and latency based experiments, we report values for 85% of the runs from the execution set, which prunes some outliers that can greatly skew results and thus, hide the important insights from a particular experiment. These outliers are typically introduced by (1) a serial launch of domains causing the first few readings to show non-shared timings for certain domains, and (2) some domains completing their runs earlier due to higher priority and/or because the launch pattern causes the last few readings for the remaining domains to again be during the unshared period. Hence, all throughput and latency graphs represent the distribution of values across the runs, with a box in the graph representing 50% of the samples around the median (or h-spread) and the lower and upper whiskers encapsulating 85% of the readings, with the minimum and maximum quantities as delimiters. It is difficult, however, to synchronize the launches of domains' GPU kernels with the execution of their threads on CPUs, leading to different orderings of CUDA calls in each run. Hence, to show cumulative performance over the entire experiment, for some experimental results, we also show the 'work done' over all of

[1] http://mathworld.wolfram.com/H-Spread.html

the runs.

Scheduling is needed when sharing accelerators: Figure 3(c) shows the overhead of sharing the GPU when applications are run both in Dom0 and in virtualized guests. In the figure, the 1VM quantities refer to overhead (or speedup) seen by a benchmark running in 1VM vs. when it is run nonvirtualized in Dom0. 2dom0 and 2VM values are similarly normalized with respect to the Dom0 values. 2dom0 values indicate execution times observed for a benchmark when it shares a GPU running in Dom0, i.e., in the absence of GPU virtualization, and 2VM values indicate similar values when run in two guest VMs sharing the GPU. For the 2VM case, the Backend implements RR, a scheduling policy that is completely fair to both VMs, and their CPU credits are set to 256 for equal CPU sharing. These measurements show that (1) while the performance seen by applications suffers from sharing (due to reduced accelerator access), (2) a clear benefit is derived for most benchmarks from using even a simple scheduling method for accelerator access. This is evident from the virtualized case that uses a round robin scheduler, which shows better performance compared with the nonvirtualized runs in Dom0 for most benchmarks, particularly the ones with lower numbers of CUDA call invocations. This shows that *scheduling is important to reduce contention in the NVIDIA driver* and thus helps minimize the resulting performance degradation. Measurements report Cuda Time and Total Time, which is the metric used in Figures 3(a)–(b).

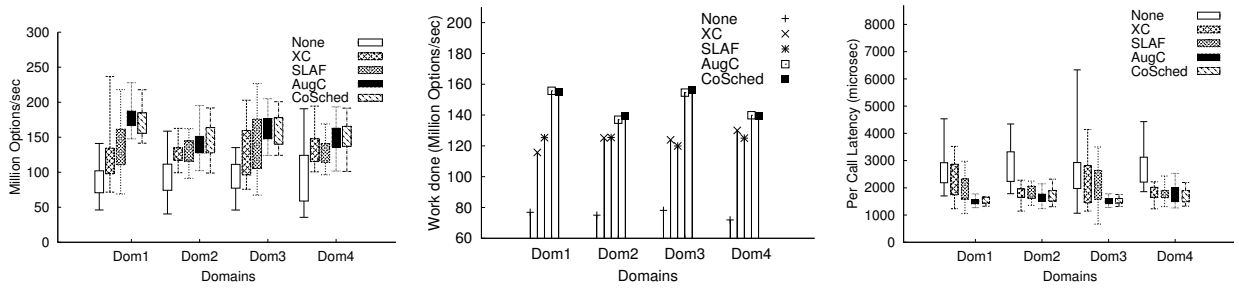
We speculate that sharing overheads could be reduced further if Pegasus was given more control over the way GPU resources are used. Additional benefits may arise from improved hardware support for sharing the accelerator, as expected for future NVIDIA hardware [27].

Coordination can improve performance: With encouraging results from the simple RR scheduler, we next experiment with the more sophisticated policies described in Section 4. In particular, we use BlackScholes (outputs options and hence its throughput is given by Options/sec) which, with more than 512 compute kernel launches and a large number of CUDA calls, has a high degree of CPU-GPU coupling. This motivates us to also report the latency numbers seen by BS.

An important insight from these experiments is that coordination in scheduling is particularly important for tightly coupled codes, as demonstrated by the fact that our base case, None, shows large variations and worse overall performance, whereas AugC and CoSched show the best performance due to their higher degrees of coordination. Figures 4(a)–(c) show that these policies perform well even when domains have equal credits. The BlackScholes run used in this experiment generates 2 million options over 512 iterations in all our domains. Figure 4(a) shows the distribution of throughput values in Million options/sec, as explained earlier. While XC and SLAF see high variation due to higher dependence on driver scheduling and no attempt for CPU and GPU coscheduling, they still perform at least 33% better than None when comparing the medians. AugC and CoSched add an additional 4%–20% improvement as seen from Figure 4(a). The higher performance seen with Dom1 and Dom3 for total work done in Figure 4(b) in case of AugC and CoSched is because of the lower signaling latency seen by the incoming and outgoing domain backend threads, due to their co-location with the scheduling thread and hence, the affected call ordering done by the NVIDIA driver (which is beyond our control).

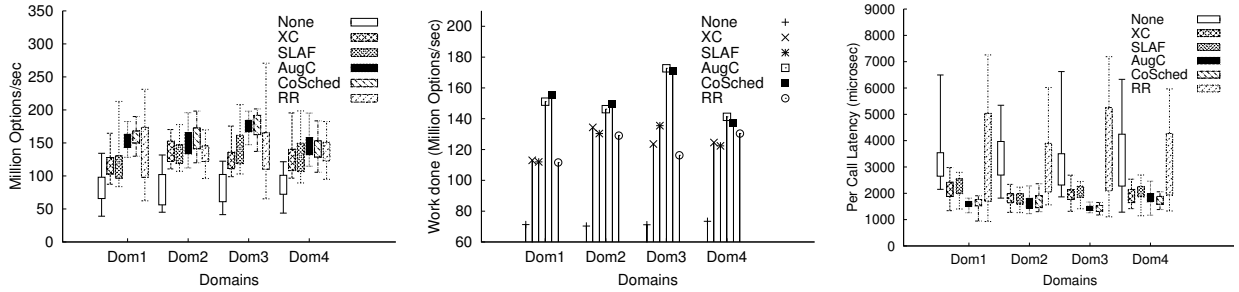
Beyond the improvements shown above, future deployment scenarios in utility data centers suggest the importance of supporting prioritization of domains. This is seen by experiments in which we modify the credits assigned to a domain, which can further improve performance (see Figure 5). We again use BlackScholes, but with Domain credits as (1) (Dom1,256), (2) (Dom2,512), (3) (Dom3,1024), and (4) (Dom4,256), respectively. The effects of such scheduling are apparent from the fact that, as shown in Figure 5(b), Dom3 succeeds in performing 2.4X or 140% more work when compared with None, with its minimum and maximum throughput values showing 3X to 2X improvement respectively. This is because domains sometimes complete early (e.g., Dom3 completes its designated runs before Dom1) which then frees up the accelerator for other domains (e.g., Dom1) to complete their work in a mode similar to non-shared operation, resulting in high throughput. The 'work done' metric captures this because average throughput is calculated for the entire application run. Another important point seen from Figure 5(c) is that the latency seen by Dom4 varies more as compared to Dom2 for say AugC because of the temporary unfairness resulting from the difference in credits between the two domains. A final interesting note is that scheduling becomes less important when accelerators are not highly utilized, as evident from other measurements not reported here.

Coordination respects proportional credit assignments: The previous experiments use equal amounts of accelerator and CPU credits, but in general, not all guest VMs need equal accelerator vs. general purpose processor resources. We demonstrate the effects of discretionary credit allocations using the BS benchmark, since it is easily configured for variable CPU and GPU execution times, based on the expected number of call and put options and the number of iterations denoted by BS⟨#options,#iterations⟩. Each domain is assigned different GPU and CPU credits denoted by Dom#⟨AccC,XC,SLA proportion⟩. This results in the configuration for this experiment being: Dom1⟨1024,256,0.2⟩ running BS⟨2mi,128⟩,



a) Throughput distribution b) Average work done c) Latency distribution

Figure 4: Performance of different scheduling schemes [BS]—equal credits for four guests



a) Throughput distribution b) Average work done c) Latency distribution

Figure 5: Performance of scheduling schemes [BS]—Credits: Dom1=256, Dom2=512, Dom3=1024, Dom4=256

Dom2⟨1024,1024,0.8⟩ running BS⟨2mi,512⟩, Dom3⟨256,1024,0.8⟩ running BS⟨0.8mi,512⟩, and Dom4⟨768,256,0.2⟩ running BS⟨1.6mi,128⟩, where mi stands for million.

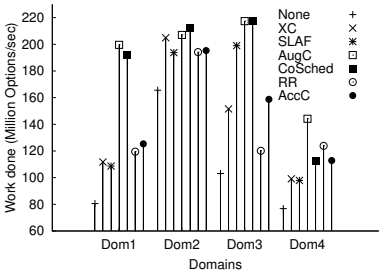
Results depicting 'total work done' in Figure 6 demonstrate that coordinated scheduling methods AugC and CoSched deal better with proportional credit assignments. Results show that domains with balanced CPU and GPU credits are more effective in getting work done—Dom2 and Dom3 (helped by high Xen credits)—than others. SLAF shows performance similar to CoSched and AugC due to its use of a feedback loop that tries to attain 80% utilization for Dom2 and Dom3 based on Xen credits. Placement of Dom4 with a high credit domain Dom2 somewhat hurts its performance, but its behavior is in accordance with its Xen credits and SLAF values, and it still sees a performance improvement of at least 18% compared to XC (lowest performance improvement among all scheduling schemes for the domain) with None. Dom1 benefits from coordination due to earlier completion of Dom3 runs, but is affected by its low CPU credits for the rest of the schemes.

One lesson from these runs is that the choice of credit assignment should be based on the expected outcome and the amount of work required by the application. How to make suitable choices is a topic for future work, particularly focusing on the runtime changes in application needs and behavior. We also realize that we cannot control the way the driver ultimately schedules requests possibly introducing high system noise and limiting achievable proportionality.

Coordination is important for latency sensitive codes: Figure 8 corroborates our earlier statement about the particular need for coordination with latency-intolerant codes. When FWT is run in all domains, first with equal CPU and GPU credits, then with different CPU credits per domain, it is apparent that 'None' (no scheduling) is inappropriate. Specifically, as seen in Figure 8, all scheduling schemes see much lower latencies and latency variations than None. Another interesting point is that the latencies seen for Dom2 and Dom3 are almost equal, despite a big difference in their credit values, for all schemes except RR (which ignores credits). This is because latencies are reduced until reaching actual virtualization overheads and thereafter, are no longer affected by differences in credits per domain. The other performance effects seen for total time can be attributed to the order in which calls reach the driver.

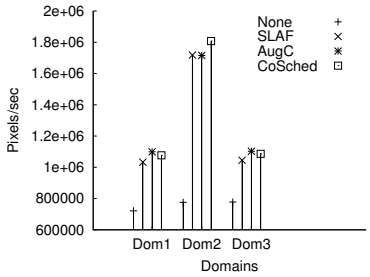
Scheduling is not always effective: There are situations in which scheduling is not effective. We have observed this when a workload is very short lived or when it shows a high degree of variation, as shown in Figure 9. These variations can be attributed to driver processing, with evidence for this attribution being that the same variability is observed in the absence of Pegasus, as seen from the 'NoVirt' bars in the figure. An idea for future work with Pegasus is to explicitly evaluate this via runtime monitoring, to establish and track penalties due to sharing, in order to then adjust scheduling to avoid such penalties whenever possible.

Scheduling does not affect performance in the absence of sharing; scheduling overheads are low: When

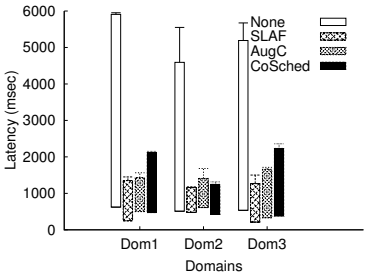


Different Xen and accelerator credits for domains

Figure 6: Performance of different scheduling schemes [BS]

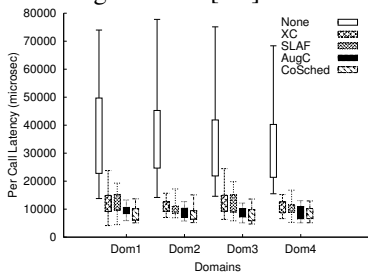


a) PicSer Work done

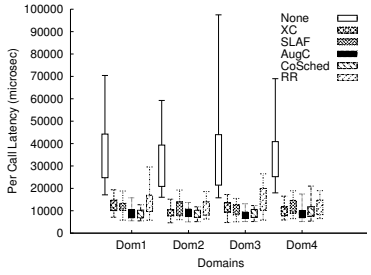


b) Latency for GPU processing

Figure 7: Performance of selected scheduling schemes for real world benchmark



a) All Doms = 256



b) Dom1=256,Dom2=1024,Dom3=2048,Dom4=256

Figure 8: Average latencies seen for [FWT]

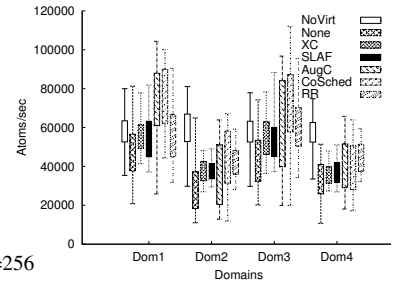


Figure 9: [CP] with sharing

using two, three, and four domains assigned equal credits, with a mix of different workloads, our measurements show that in general, scheduling works well and exhibits little variation, especially in the absence of accelerator sharing. While those results are omitted due to lack of space, we do report the worst case scheduling overheads seen per scheduler call in Table 2, for different scheduling policies. MS in the table refers to the Monitor and Sweep thread responsible for monitoring credit value changes for guest VMs and cleaning out state for non-existing VMs. Xen kernel refers to the changes made to the hypervisor CPU scheduling method. Acc0 and Acc1 refer to the schedulers (for timer based schemes like RR, XC, SLAF) in our dual accelerator testbed. Hype refers to the user level thread run for policies like AugC and CoSched for coordinating CPU and GPU activities.

As seen from the table, the Pegasus backend components have low overhead. For example, XC sees ∼0.5ms per scheduler call per accelerator, compared to a typical execution time of CUDA applications of between 250ms to 5000ms and with typical scheduling periods of 30ms. The most expensive component, with an overhead of ∼1ms, is MS, which runs once every second.

Scheduling complex workloads: When evaluating scheduling policies with the PicSer application, we run three dual-core, 512MB guests on our testbed. One VM (Dom2) is used for priority service and hence given 1024 credits and 1 GPU, while the remaining two are assigned 256 credits, and they share the second GPU. VM2 is latency-sensitive, and all of the VMs care about throughput. Scheduling is important because CPUs are shared by multiple VMs. Figure 7(a) shows the average throughput (Pixels/sec to incorporate different image sizes) seen by each VM with four different policies. We choose AugC and CoSched to highlight the co-scheduling differences. None is to provide a baseline, and SLAF is an enhanced version of all of the credit based schemes. AugC tries to improve the throughput of all VMs, which results in a somewhat lower value for Dom2. CoSched gives priority to Dom2 and can penalize other VMs, as evident from the GPU latencies shown in Figure 7(b). 'No scheduling' does not perform well. More generally, it is clear that coordinated scheduling can be effective in meeting the requirements of multi-VM applications sharing CPU and GPU resources.

Policy	MS (μsec)	Xen Kernel (μsec)	Acc0/Hype (μsec)	Acc1 (μsec)
None	272	0.85	0	0
XC	1119	0.85	507	496
AugC	1395	0.9	3.36	0
SLAF	1101	0.95	440	471
CoSched	1358	0.825	2.71	0

Table 2: Backend scheduler overhead

6.4 Discussion

Experimental results show that the Pegasus approach efficiently virtualizes GPUs and in addition, can effectively schedule their use. Even basic accelerator request scheduling can improve sharing performance, with additional benefits derived from active scheduling coordi-

nation schemes. Among these methods, XC can perform quite well, but fails to capitalize on CPU-GPU coordination opportunities for tightly coupled benchmarks. SLAF, when applied to CPU credits, has a smoothing effect on the high variations of XC, because of its feedback loop. For most benchmarks, especially those with a high degree of coupling, AugC and CoSched perform significantly better that other schemes, but require small changes to the hypervisor. More generally, scheduling schemes work well in the absence of over-subscription, helping regulate the flow of calls to the GPU. Regulation also results in lowering the degrees of variability caused by un-coordinated use of the NVIDIA driver.

AugC and CoSched, in particular, constitute an interesting path toward realizing our goal of making accelerators first class citizens, and further improvements to those schemes can be derived from gathering additional information about accelerator resources. There is not, however, a single 'best' scheduling policy. Instead, there is a clear need for diverse policies geared to match different system goals and to account for different application characteristics.

Pegasus scheduling uses global platform knowledge available at hypervisor level, and its implementation benefits from hypervisor-level efficiencies in terms of resource access and control. As a result, it directly addresses enterprise and cloud computing systems in which virtualization is prevalent. Yet, clearly, methods like those in Pegasus can also be realized at OS level, particularly for the high performance domain where hypervisors are not yet in common use. In fact, we are currently constructing a CUDA interposer library for non-virtualized, native guest OSes, which we intend to use to deploy scheduling solutions akin to those realized in Pegasus at large scale on the Keeneland machine.

7 Related Work

The importance of dealing with the heterogeneity of future multi-core platforms is widely recognized. Cypress [10] has expressed the design principles for hypervisors actually realized in Pegasus (e.g., partitioning, localization, and customization), but Pegasus also articulates and evaluates the notion of coordinated scheduling. Multikernel [4] and Helios [26] change system structures for multicores, advocating distributed system models and satellite kernels for processor groups, respectively. In comparison, Pegasus retains the existing operating system stack, then uses virtualization to adapt to diverse underlying hardware, and finally, leverages the federation approach shown scalable in other contexts to deal with multiple resource domains.

Prior work on GPU virtualization has used the OpenGL API [21] or 2D-3D graphics virtualization (DirectX, SVGA) [9]. In comparison, Pegasus operates on entire computational kernels more readily co-scheduled with VCPUs running on general purpose CPUs. This approach to GPU virtualization is outlined in an earlier workshop paper, termed GViM [13], which also presents some examples that motivate the need for QoS-aware scheduling. In comparison, this paper thoroughly evaluates the approach, develops and explores at length the notion of coordinated scheduling and the scheduling methods we have found suitable for GPGPU use and for latency- vs. throughput-intensive enterprise codes.

While similar in concept, Pegasus differs from coordinated scheduling at the data center level, in that its deterministic methods with predictable behavior are more appropriate at the fine-grained hypervisor level than the loosely-coordinated control-theoretic or statistical techniques used in data center control [20]. Pegasus co-scheduling differs in implementation from traditional gang scheduling [36] in that (1) it operates across multiple scheduling domains, i.e., GPU vs. CPU scheduling, without direct control over how each of those domains schedules its resources, and (2) because it limits the idling of GPUs, by running workloads from other aVCPUs when a currently scheduled VCPU does not have any aVCPUs to run. This is appropriate because Pegasus co-scheduling schemes can afford some skew between CPU and GPU components, since their aim is not to solve the traditional locking issue.

Recent efforts like Qilin [23] and predictive runtime code scheduling [16] both aim to better distribute tasks across CPUs and GPUs. Such work is complementary and could be used combined with the runtime scheduling methods of Pegasus. Upcoming hardware support for accelerator-level contexts, context isolation, and context-switching [27] may help in terms of load balancing opportunities and more importantly, it will help improve accelerator sharing [9].

8 Conclusions and Future Work

This paper advocates making all of the diverse cores of heterogeneous manycore systems into first class schedulable entities. The Pegasus virtualization-based approach for doing so, is to abstract accelerator interfaces through virtualization and then devise scheduling methods that coordinate accelerator use with that of general purpose host cores. The approach is applied to a combined NVIDIA- and x86-based GPGPU multicore prototype, enabling multiple guest VMs to efficiently share heterogenous platform resources. Evaluations using a large set of representative GPGPU benchmarks and computationally intensive web applications result in insights that include: (1) the need of coordination when sharing accelerator resources, (2) its critical importance for applications that frequently interact across the CPU-GPU boundary, and (3) the need for diverse policies when co-

ordinating the resource management decisions made for general purpose vs. accelerator cores.

Certain elements of Pegasus remain under development and/or are subject of future work. Admission control methods can help alleviate certain problems with accelerator sharing, such as those caused by insufficient accelerator resources (e.g., memory). Runtime load balancing across multiple accelerators would make it easier to deal with cases in which GPU codes do not perform well when sharing accelerator resources. Static profiling and runtime monitoring could help identify such codes. There will be some limitations in load balancing, however, because of the prohibitive costs in moving the large amounts of memory allocated on completely isolated GPU resources. This restricts load migration to cases in which the domain has no or little state on a GPU. As a result, the first steps in our future work will be to provide Pegasus scheduling methods with additional options for accelerator mappings and scheduling, by generalizing our implementation to use both local and non-local accelerators (e.g., when they are connected via high end network links like Infiniband). Despite these shortcomings, the current implementation of Pegasus not only enables multiple VMs to efficiently share accelerator resources, but also achieves considerable performance gains with its coordination methods.

Acknowledgments

We would like to thank our shepherd Mr. Muli Ben-Yehuda and other anonymous reviewers for their insight on improving the paper. We would also like to thank Priyanka Tembey, Romain Cledat and Tushar Kumar for their feedback and assistance with the paper.

References

[1] AMAZON INC. High Performance Computing Using Amazon EC2. http://aws.amazon.com/ec2/hpc-applications/.

[2] BAKHODA, A., YUAN, G. L., FUNG, W. W., ET AL. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS* (Boston, USA, 2009).

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., ET AL. Xen and the art of virtualization. In *SOSP* (Bolton Landing, USA, 2003).

[4] BAUMANN, A., BARHAM, P., DAGAND, P. E., ET AL. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP* (Big Sky, USA, 2009).

[5] BERGMANN, A. The Cell Processor Programming Model. In *LinuxTag* (2005).

[6] BORDAWEKAR, R., BONDHUGULA, U., AND RAO, R. Believe It or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application! Tech. Report RC24982, IBM T. J. Watson Research Center,, 2010.

[7] CHISNALL, D. *The Definitive Guide to the Xen Hypervisor*, 1st ed. Prentice Hall, 2008.

[8] DIAMOS, G., AND YALAMANCHILI, S. Harmony: An Execution Model and Runtime for Heterogeneous Many Core Systems. In *HPDC Hot Topics* (Boston, USA, 2008).

[9] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. In *WIOV* (San Diego, USA, 2008).

[10] FEDOROVA, A., KUMAR, V., KAZEMPOUR, V., ET AL. Cypress: A Scheduling Infrastructure for a Many-Core Hypervisor. In *MMCS* (Boston, USA, 2008).

[11] GOVIL, K., TEODOSIU, D., HUANG, Y., ET AL. Cellular Disco: resource management using virtual clusters on shared-memory multiprocessors. In *SOSP* (Charleston, USA, 1999).

[12] GUEVARA, M., GREGG, C., HAZELWOOD, K., ET AL. Enabling Task Parallelism in the CUDA Scheduler. In *PMEA* (Raleigh, USA, 2009).

[13] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., ET AL. GViM: GPU-accelerated Virtual Machines. In *HPCVirt* (Nuremberg, Germany, 2009).

[14] GUPTA, V., XENIDIS, J., TEMBEY, P., ET AL. Cellule: Lightweight Execution Environment for Accelerator-based Systems. Tech. Rep. GIT-CERCS-10-03, Georgia Tech, 2010.

[15] HEINIG, A., STRUNK, J., REHM, W., ET AL. *ACCFS - Operating System Integration of Computational Accelerators Using a VFS Approach*, vol. 5453. Springer Berlin, 2009.

[16] JIMÉNEZ, V. J., VILANOVA, L., GELADO, I., ET AL. Predictive Runtime Code Scheduling for Heterogeneous Architectures. In *HiPEAC* (Paphos, Cyprus, 2009).

[17] JOHNSON, C., ALLEN, D. H., BROWN, J., ET AL. A Wire-Speed PowerTM Processor: 2.3GHz 45nm SOI with 16 Cores and 64 Threads. In *ISSCC* (San Francisco, USA, 2010).

[18] KERR, A., DIAMOS, G., AND YALAMANCHILI, S. A Characterization and Analysis of PTX Kernels. In *IISWC* (Austin, USA, 2009).

[19] KHRONOS GROUP. The OpenCL Specification. http://tinyurl.com/OpenCL08, 2008.

[20] KUMAR, S., TALWAR, V., KUMAR, V., ET AL. vManage: Loosely Coupled Platform and Virtualization Management in Data Centers. In *ICAC* (Barcelona, Spain, 2009).

[21] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., ET AL. VMM-independent graphics acceleration. In *VEE* (San Diego, CA, 2007).

[22] LANGE, J., PEDRETTI, K., DINDA, P., ET AL. Palacios: A New Open Source Virtual Machine Monitor for Scalable High Performance Computing. In *IPDPS* (Atlanta, USA, 2010).

[23] LUK, C.-K., HONG, S., AND KIM, H. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Micro-42* (New York, USA, 2009).

[24] MARCIAL, E. The ICE Financial Application. http://www.theice.com, 2010. Private Communication.

[25] MICROSOFT CORP. What is Photosynth? http://photosynth.net/about.aspx, 2010.

[26] NIGHTINGALE, E. B., HODSON, O., MCILROY, R., ET AL. Helios: heterogeneous multiprocessing with satellite kernels. In *SOSP* (Big Sky, USA, 2009).

[27] NVIDIA CORP. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. http://tinyurl.com/nvidia-fermi-whitepaper.

[28] NVIDIA CORP. NVIDIA CUDA Compute Unified Device Architecture. http://tinyurl.com/cx3tl3, 2007.

[29] RAJ, H., AND SCHWAN, K. High performance and scalable I/O virtualization via self-virtualized devices. In *HPDC* (Monterey, USA, 2007).

[30] RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., ET AL. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP* (Salt Lake City, USA, 2008).

[31] SHIMPI, A. L. Intel's Sandy Bridge Architecture Exposed. http://tinyurl.com/SandyBridgeArch.

[32] SNAPFISH. About Snapfish. http://www.snapfish.com.

[33] SNAVELY, N., SEITZ, S. M., AND SZELISKI, R. Modeling the World from Internet Photo Collections. *International Journal of Computer Vision 80*, 2 (2008).

[34] TURNER, J. A. The Los Alamos Roadrunner Petascale Hybrid Supercomputer: Overview of Applications, Results, and Programming. Roadrunner Technical Seminar Series, 2008.

[35] VETTER, J., GLASSBROOK, D., DONGARRA, J., ET AL. Keeneland - Enabling Heterogeneous Computing For The Open Science Community. http://tinyurl.com/KeenelandSC10, 2010.

[36] VMWARE CORP. VMware vSphere 4: The CPU Scheduler in VMware ESX 4. http://tinyurl.com/ykenbjw, 2009.

vIC: Interrupt Coalescing for Virtual Machine Storage Device IO

Irfan Ahmad *Ajay Gulati* *Ali Mashtizadeh*
{irfan, agulati, ali}@vmware.com
VMware, Inc., Palo Alto, CA 94304

Abstract

Interrupt coalescing is a well known and proven technique for reducing CPU utilization when processing high IO rates in network and storage controllers. Virtualization introduces a layer of virtual hardware for the guest operating system, whose interrupt rate can be controlled by the hypervisor. Unfortunately, existing techniques based on high-resolution timers are not practical for virtual devices, due to their large overhead. In this paper, we present the design and implementation of a virtual interrupt coalescing (vIC) scheme for virtual SCSI hardware controllers in a hypervisor.

We use the number of *commands in flight* from the guest as well as the current *IO rate* to dynamically set the degree of interrupt coalescing. Compared to existing techniques in hardware, our work does not rely on high-resolution interrupt-delay timers and thus leads to a very efficient implementation in a hypervisor. Furthermore, our technique is generic and therefore applicable to all types of hardware storage IO controllers which, unlike networking, don't receive anonymous traffic. We also propose an optimization to reduce inter-processor interrupts (IPIs) resulting in better application performance during periods of high IO activity. Our implementation of virtual interrupt coalescing has been shipping with VMware ESX since 2009. We present our evaluation showing performance improvements in micro benchmarks of up to 18% and in TPC-C of up to 5%.

1 Introduction

The performance overhead of virtualization has decreased steadily in the last decade due to improved hardware support for hypervisors. This and other storage device optimizations have led to increasing deployments of IO intensive applications on virtualized hosts. Many important enterprise applications today exhibit high IO rates. For example, transaction processing loads can issue hundreds of very small IO operations in parallel resulting in tens of thousands of IOs per second (IOPS). Such high IOPS are now within reach of even more IT organizations with faster storage controllers, wider adoption of solid-state disks (SSDs) as front-end tier in storage arrays and increasing deployments of high performance consolidated storage devices using Storage Area Network (SAN) or Network-Attached Storage (NAS) protocols.

For high IO rates, the CPU overhead for handling all the interrupts can get very high and eventually lead to lack of CPU resources for the application itself [7, 14]. CPU overhead is even more of a problem in virtualization scenarios where we are trying to consolidate as many virtual machines into one physical box as possible. Freeing up CPU resources from one virtual machine (VM) will improve performance of other VMs on the same host. Traditionally, interrupt coalescing or moderation has been used in network and storage controller cards to limit the number of times that application execution is interrupted by the device to handle IO completions. Such coalescing techniques have to carefully balance an increase in IO latency with the improved execution efficiency due to fewer interrupts.

In hardware controllers, fine-grained timers are used in conjunction with interrupt coalescing to keep an upper bound on the latency of IO completion notifications. Such timers are inefficient to use in a hypervisor and one has to resort to other pieces of information to avoid longer delays. This problem is challenging for several other reasons, including the desire to maintain a small code size thus keeping the trusted computing base to a manageable size. We treat the virtual machine workload as unmodifiable and as an opaque black box. We also assume based on earlier work that guest workloads can change their behavior very quickly [6, 10].

In this paper, we target the problem of coalescing interrupts for virtual devices without assuming any support from hardware controllers and without using high res-

olution timers. Traditionally, there are two parameters that need to be balanced: maximum interrupt delivery latency (MIDL) and maximum coalesce count (MCC). The first parameter denotes the maximum time to wait before sending the interrupt and the second parameter denotes the number of accumulated completions before sending an interrupt to the operating system (OS). The OS is interrupted based on whichever parameter is hit first.

We propose a novel scheme to control for both MIDL and MCC implicitly by setting the *delivery ratio* of interrupts based on the current number of commands in flight (CIF) from the guest OS and overall IO completion rate. The ratio, denoted as R, is simply the ratio of how many virtual interrupts are sent to the guest divided by the number of actual IO completions received by the hypervisor on behalf of that guest. Note that $0 < R \leq 1$. Lower values of delivery ratio, R, denotes a higher degree of coalescing. We increase R when CIF is low and decrease the delivery rate R for higher values of CIF.

The key insight in the paper is that unlike network IO, CIF can be used directly for storage controllers because each request has a corresponding command in flight prior to completion. Also, based on the characteristics of storage devices, it is important to maintain certain number of commands in flight to efficiently utilize the underlying storage device [9, 11, 23]. The benefits of command queuing are well known and concurrent IOs are used in most storage arrays to maintain high utilization. Another challenge in coalescing interrupts for storage IO requests is that many important applications issue synchronous IOs. Delaying the completion of prior IOs can delay the issue of future ones, so one has to be very careful about minimizing the latency increase.

Another problem we address is specific to hypervisors, where the host storage stack has to receive and process an IO completion before routing it to the issuing VM. The hypervisor may need to send inter-processor interrupts (IPIs) from the CPU that received the hardware interrupt to the remote CPU where the VM is running for notification purposes. We provide an optimization to reduce the number of IPIs issued using the timestamp of the last interrupt that was sent to the guest OS. This reduces the overall number of IPIs while bounding the latency of notifying the guest OS about an IO completion.

We have implemented our virtual interrupt coalescing (vIC) techniques in the VMware ESX hypervisor [21] though they can be applied to any hypervisor including type 1 and type 2 as well as hardware storage controllers. Experimentation with a set of micro benchmarks shows that vIC techniques can improve both workload throughput and CPU overheads related to IO processing by up to 18%. We also evaluated vIC against the TPC-C workload and found improvements of up to 5%. The vIC implementation discussed here is being used by thousands of customers in the currently shipping ESX version.

The next section presents background on VMware ESX Server architecture and overall system model along with a more precise problem definition. Section 3 presents the design of our virtual interrupt coalescing mechanism along with a discussion of some practical concerns. An extensive evaluation of our implementation is presented in Section 4, followed by some lessons learned from our deployment experience in real world in Section 5. Section 6 presents an overview of related work followed by conclusions and directions for future work in Sections 7 and 8 respectively.

2 System Model

Our system model consists of two components in the VMware ESX hypervisor: VMkernel and the virtual machine monitor (VMM). The VMkernel is a hypervisor kernel, a thin layer of software controlling access to physical resources among virtual machines. The VMkernel provides isolation and resource allocation among virtual machines running on top of it. The VMM is responsible for correct and efficient virtualization of the *x*86 instruction set architecture as well as emulation of high performance virtual devices. It is also the conceptual equivalent of a "process" to the ESX VMkernel. The VMM intercepts all the privileged operations from a VM including IO and handles them in cooperation with the VMkernel.

Figure 1 shows the ESX VMkernel executing storage stack code on the CPU on the right and an example VM running on top of its virtual machine monitor (VMM) running on the left processor. In the figure, when an interrupt is received from a storage adapter (1), appropriate code in the VMkernel is executed to handle the IO completion (2) all the way up to the vSCSI subsystem which narrows the IO to a specific VM. Each VMM shares a common memory area with the ESX VMkernel, where the VMkernel posts IO completions in a queue (3) following which it may issue an inter-process interrupt or IPI (4) to notify the VMM. The VMM can pick up the completions on its next execution (5) and process them (6) resulting finally in the virtual interrupt being fired (7).

Without explicit interrupt coalescing, the VMM always asserts the level-triggered interrupt line for every IO. Level-triggered lines do some implicit coalescing already but that only helps if two IOs are completed back-to-back in the very short time window before the guest interrupt service routine has had the chance to deassert the line.

Only the VMM can assert the virtual interrupt line and it is possible after step 3 that the VMM may not get a chance to execute for a while. To limit any latency implications of a VM not entering into the VMM, the

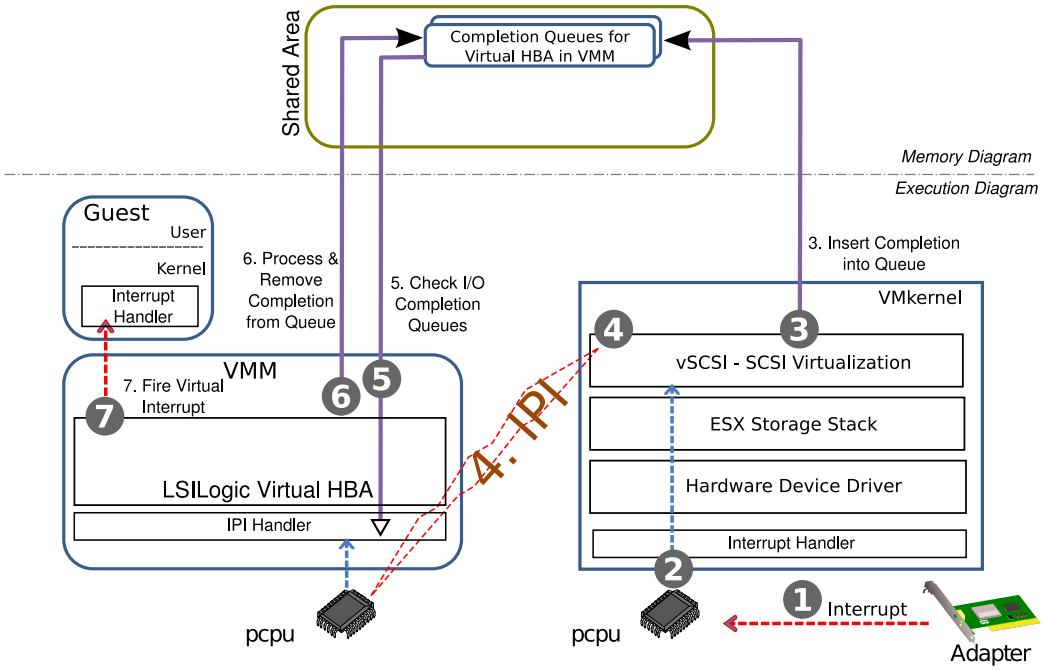


Figure 1: Virtual Interrupt Delivery Mechanism. When a disk IO completes, an interrupt is fired (1) from a physical adapter to a particular Physical CPU (PCPU) where the interrupt handler of the hypervisor delivers it to the appropriate device driver (2). Higher layers of the hypervisor storage stack process the completion until the IO is matched (vSCSI layer) to a particular Guest Operating System which issued the IO and its corresponding Virtual Machine Monitor (VMM). vSCSI then updates the shared completion queue for the VMM (3) and if the guest or VMM is currently executing, issues an inter-processor interrupt (IPI) to the target PCPU where the VMM is known to be running (4). The IPI is only a latency optimization since the VMM would have inspected the shared queues the next time the guest exited to the VMM anyway. The remote VMM's IPI handler takes the signal and (5) inspects the completion queues of its virtual SCSI host bus adapters (HBAs), processes and virtualizes the completions (6) and fires a virtual interrupt to be handled by the guest (7).

VMkernel will take one of two actions. It will schedule the VM if it is descheduled. Otherwise, if both the VM and the VMkernel are executing on separate cores at the same time, the VMkernel sends an IPI, in step 4 in the figure. This IPI is purely an optimization to provide lower latency IO completions to the guest. Without the IPI, guests may execute user level code for an extended period without triggering any hypervisor intercept that would allow for virtual interrupt delivery. Correctness guarantees can still be met even if the IPI isn't issued since the VMM will pickup the completion as a matter of course the next time that it gets invoked via a timer interrupt or a guest exiting into VMM mode due to a privileged operation.

Based on the design described above, there are two inefficiencies in the existing mechanism. First the VMM will interrupt the guest for every completion that it sees posted by the VMkernel. We would like to coalesce these to reduce the guest CPU overhead during high IO rates. Second, IPIs are very costly and are used mainly as a latency optimization. It would be desirable to dramatically reduce them if one could track the rate at which completions are being picked up by the VMM. All this needs to be done without the help of fine grained timers because they are prohibitively expensive in a hypervisor. Thus the main challenges in coalescing virtual interrupts can be summarized as:

1. How to control the rate of interrupt delivery from a VMM to a guest without loss of throughput?
2. How and when to delay the IPIs without inducing high IO latencies?

In the next section, we present our virtual interrupt coalescing mechanisms to efficiently resolve both of these challenges.

3 vIC Design

In this section, we first present some background on existing coalescing mechanisms and explain why they can-

not be used in our environment. Next, we present our approach at a high level followed by the details of each component and a discussion of specific implementation issues.

3.1 Background

When implemented in physical hardware controllers, interrupt coalescing generally makes use of high resolution timers to cap the amount of extra latency that interrupt coalescing might introduce. Such timers allow the controllers to directly control MIDL (maximum interrupt delivery latency) and adapt MCC (maximum coalesce count) based on the current rate. For example, one can configure MCC based on a recent estimate of interrupt arrivals and put a hard cap on latency by using high resolution timers to control MIDL. Some devices are known to allow a configurable MIDL in increments of tens of microseconds.

Such high resolution timers are generally used in dedicated IO processors where the firmware timer handler overhead can be well contained and the hardware resources can be provisioned at design time to meet the overhead constraints. However, in any general purpose operating system or hypervisor, it is generally not considered feasible to program high resolution timing as a matter of course. The associated CPU overhead is simply too high.

If we were to try to directly map that MCC/MIDL solution to virtual interrupts, we would be forced to drive the system timer interrupt using resolutions as high as 100 μs. Such a high interrupt rate would have prohibitive performance impact on the overall system both in terms of the sheer CPU cost of running the software interrupt handler ten thousand times a second, as well as the first- and second-order context switching overhead associated with each of them. As a comparison, Microsoft Windows 7 typically sets up its timers to go off every 15.6 *ms* or down to 1 *ms* in special cases whereas VMware ESX configures timers in the range of 1 *ms* and 10 *ms* or even longer when using one-shot timers. This is orders of magnitude lower resolution than what is used by typical storage hardware controller firmware.

3.2 Our approach

In our design, we define a parameter called interrupt delivery ratio *R*, as the ratio of interrupts delivered to the guest and the actual number of interrupts received from the device for that guest. A lower delivery ratio implies a higher degree of coalescing. We dynamically set our interrupt delivery ratio, *R*, in a way that will provide coalescing benefits for CPU efficiency as well as tightly control any extra vIC-related latency. This is done using commands in flight (CIF) as the main parameter and IO completion rate (measured as IOs per sec or IOPS) as a secondary control.

At a high level, if IOPS is high, we can coalesce more interrupts within the same time period, thereby improving CPU efficiency. However, we still want to avoid and limit the increase in latency for cases when the IOPS changes drastically or when the number of issued commands is very low. SSDs can typically do tens of thousands of IOPS even with CIF = 1, but delaying IOs in this case would hurt overall performance.

To control IO delay, we use CIF as a guiding parameter, which determines the overall impact that the coalescing can have on the workload. For example, coalescing 4 IO completions out of 32 outstanding might not be a problem since we are able to keep the storage device busy with the remaining 28, whereas even a slight delay caused by coalescing 2 IOs out of 4 outstanding could result in the resources of the storage device not getting fully utilized. Thus we want to vary the delivery ratio *R* in inverse proportion of the CIF value. Using both CIF values and estimated IOPS value, we are able to provide effective coalescing for a wide variety of workloads.

There are three main parameters used in our algorithm:

- *iopsThreshold*: IOPS value below which no interrupt coalescing is done.
- *cifThreshold*: CIF value below which no interrupt coalescing is done.
- *epochPeriod*: Time interval after which we re-evaluate the delivery ratio, in order to react to the change in the VM workload.

The algorithm operates in one of the three modes:
(1) **Low-IOPS ($R = 1$):** We turn off vIC if the achieved throughput of a workload ever drops below the *iopsThreshold*. Recall that we do not have a high resolution timer. If we did, whenever it would fire, it would allow us to determine if we have held on to an IO completion for too long. A key insight for us is that instead of a timer, we can actually rely on *future* IO-completion events to give our code a chance to control extra latency.

For example, an IOPS value of 20,000 means that on average there will be a completion returned every 50 μs. Our default iopsThreshold is 2000 that implies a completion on average every 500 μs. Therefore, at worst, we can add that amount of latency. For higher IOPS, the extra latency only decreases. In order to do this, we keep an estimate of the current number of IOPS completed by the VM.

(2) **Low-CIF ($R = 1$):** We turn off vIC whenever the number of outstanding IOs (CIF) drops below a configurable parameter *cifThreshold*. Our interrupt coalescing

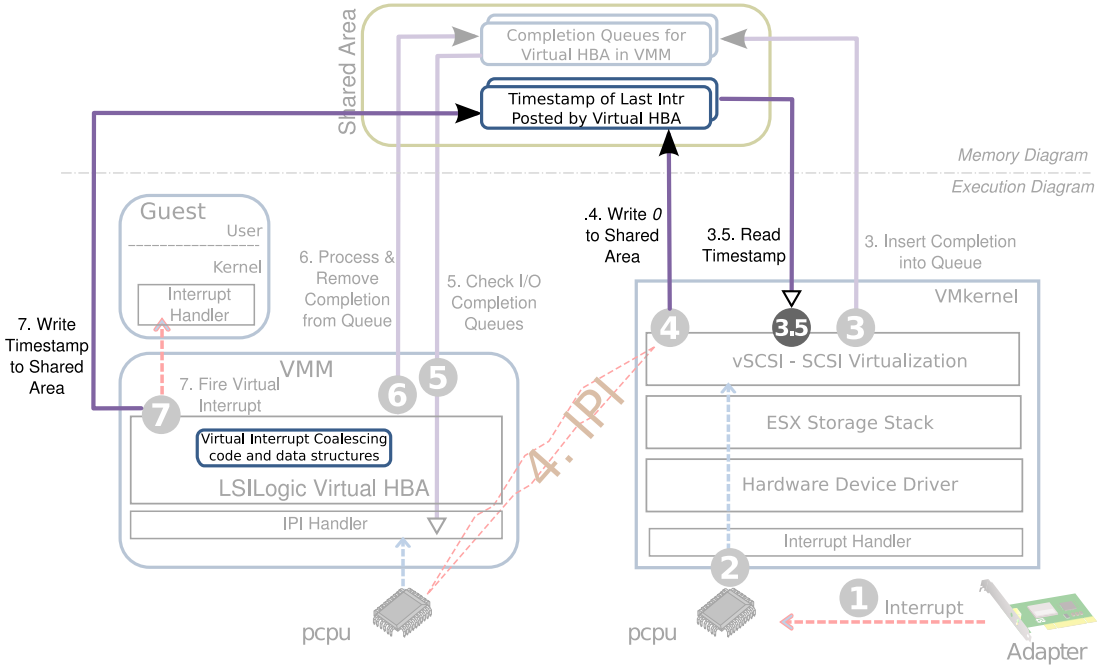


Figure 2: Virtual Interrupt Delivery Steps. In addition to Figure 1, vIC adds a new shared area object tracking the last time that the VMM fired an interrupt. Before sending the IPI, vSCSI checks to ensure that time since the last VMM-induced virtual interrupt is less than a configurable threshold. If not so, an IPI is still fired, otherwise, it is deferred. In the VMM, an interrupt coalescing scheme is introduced. Note that we did not introduce a high-resolution timer and instead rely on the subsequent IO completions themselves to drive the vIC logic and to regulate the vIC related delay.

algorithm tries to be very conservative so as to not increase the application IO latency for trickle IO workloads. Such workloads have very strong IO inter dependencies and generally issue only a very small number of outstanding IOs.

A canonical example of an affected workload is `dd`, which issues *one* IO at a time. For `dd`, if we had coalesced an interrupt, it would actually hang forever. In fact, waiting is completely useless for such cases and it only adds extra latency. When only a small number of IOs (cifThreshold) remain outstanding on an adapter, we stop coalescing. Otherwise, there may be a throughput reduction because we are delaying a large percentage of IOs.

(3) **Variable *R* based on CIF:** Setting the delivery ratio (R) dynamically is challenging since we have to balance the CPU efficiency gained by coalescing against additional latency that may be added especially since that may in turn lower the achieved throughput. We discuss our computation of R next.

3.2.1 Dynamic Adjustment of Delivery Ratio *R*

Which ratio is picked depends upon the number of commands in flight (CIF) and the configuration option "cifThreshold". As CIF increases, we have more room to coalesce. For workloads with multiple outstanding IOs, the extra delay works well since they are able to amortize the cost of the interrupt being delivered to process more than one IO. For example, if the CIF value is 24, even if we coalesce 3 IOs at a time, the application will have 21 other IOs pending at the storage device to keep it busy.

In deciding the value of R, we have two main issues to resolve. First we cannot choose an arbitrary fractional value of R and implement that because of the lack of floating point calculations in the VMM code. Second, a simple ratio of the form $1/x$ based on a counter x would imply that the only delivery-ratio options available to the algorithm would be (100%, 50%, 25%, 12.5%, ...). The jump from 100% down to 50% is actually too drastic. Instead, we found that to be able to handle a multitude of situations, we need to have delivery ratios, anywhere from 100% down to 6.25%. In order to do this we chose to set *two* fields, countUp and skipUp, dynamically to express the delivery ratios. Intuitively, we deliver (countUp) out of every (skipUp) interrupts, *i.e.* $R = countUp/skipUp$. For example, to deliver 80% of the interrupts, countUp = 4 and skipUp = 5 whereas for 6.25% countUp = 1 and skipUp = 16. Table 1 shows the full range of values as encoded in Algorithm 1. By allowing ratios between 100% and 50%, we can tightly

Algorithm 1: Delivery Ratio Determination

```
IntrCoalesceRecalc(int cif)
currIOPS : Current throughput in IOs per sec;
cif : Current # of commands in flight (CIF);
cifThreshold : Configurable min CIF (default=4);
if currIOPS < iopsThreshold ∨ cif < cifThreshold then
    /* R = 1 */
    countUp ⟵ 1;
    skipUp ⟵ 1;
else if cif < 2 ∗ cifThreshold then
    /* R = 0.8 */
    countUp ⟵ 4;
    skipUp ⟵ 5;
else if cif < 3 ∗ cifThreshold then
    /* R = 0.75 */
    countUp ⟵ 3;
    skipUp ⟵ 4;
else if cif < 4 ∗ cifThreshold then
    /* R = 0.66 */
    countUp ⟵ 2;
    skipUp ⟵ 3;
else
    /* R = 8/CIF */
    countUp ⟵ 1;
    skipUp ⟵ cif/(2 ∗ cifThreshold);
```

Algorithm 2: VMM—IO Completion Handler

```
cif : Current # of commands in flight (CIF);
cifThreshold : Configurable min CIF (default=4);
epochStart : Time at start of current epoch (global);
epochPeriod : Duration of each epoch (global);
diff ⟵ currTime() − epochStart;
if diff > epochPeriod then
    IntrCoalesceRecalc(cif);
if cif < cifThreshold then
    counter ⟵ 1;
    deliverIntr();
else if counter < countUp then
    counter++;
    deliverIntr();
else if counter >= skipUp then
    counter ⟵ 1;
    deliverIntr();
else
    counter++;
    /* don't deliver */
if Interrupt Was Delivered then
    SharedArea.timeStamp ⟵ currTime();
```

control the throughput loss at smaller CIF.

The exact values of R are determined based on experimentation and to support the efficient implementation in a VMM. Algorithm 1 shows the exact values of delivery ratio R as a function of CIF, cifThreshold and iopsThreshold. Next we will discuss the details of interrupt delivery mechanism and some optimizations in implementing this computation.

CIF	Intr Delivery Ratio R as %
1-3	100%
4-7	80%
8-11	75%
12-15	66%
CIF $\geq$ 16	8 / CIF
e.g., CIF = 64	12%

Table 1: Default interrupt delivery ratio (R) as a function of CIF. $cifThreshold$ is set to the default of 4.

3.2.2 Delivering Interrupts

On any given IO completion, the VMM needs to decide whether to post an interrupt to the guest or to coalesce it with a future one. This decision logic is captured in pseudo code in Algorithm 2. First, at every "epoch period", which defaults to 200 *ms*, we reevaluate the vIC rate so we can react to changes in workloads. This is done in function $IntrCoalesceRecalc()$, the pseudo code for which is found in Algorithm 1.

Next, we check to see if the new CIF is below the cifThreshold. If such a condition happens, we immediately deliver the interrupt. The VMM is designed as a very high performance software system where we worry about code size (in terms of both lines of code (LoC) and bytes of `.text`). Ultimately, we have to calculate for *each* IO completion whether or not to deliver a virtual interrupt given the ratio $R = countUp/skipUp$. Since this decision is on the critical path of IO completion, we have designed a simple but very condensed logic to do so with the minimum number of LoC, which needs careful explanation.

In Algorithm 2, $counter$ is an abstract number that counts up from 1 till $countUp - 1$ delivering an interrupt each time. It then continues to count up till $skipUp - 1$ while skipping each time. Finally, once $counter$ reaches $skipUp$, it is reset back to 1 along with an interrupt delivery. Let us look at two examples of a series of $counter$ values as more IOs come in, along with whether the algorithm delivers an interrupt as tuples of ⟨counter, deliver?⟩. For $countUp/skipUp$ ratio of $3/4$, a series of IOs looks like:

⟨1, yes⟩, ⟨2, yes⟩, ⟨3, no⟩, ⟨4, yes⟩.

Whereas for $countUp/skipUp$ of $1/5$:

⟨1, no⟩, ⟨2, no⟩, ⟨3, no⟩, ⟨4, no⟩, ⟨5, yes⟩.

Next we look at the optimization related to reducing the number of IPIs sent to a VM during high IO rate.

3.3 Reducing IPIs

So far, we have described the mechanism for virtual interrupt coalescing inside the VMM. As mentioned in Section 2 and illustrated in Figure 1, another component involved in IO delivery is the ESX VMkernel. Recall that IO completions from hardware controllers are handled by this component and sent to the VMM, an operation that can require an IPI in case the guest is currently running on the remote processor. Since IPIs are expensive, we would like to avoid them or at the very least minimize their occurrence. Note that the IPI is a mechanism to force the VMM to wrest execution control away from the guest to process a completion. As such it is purely a latency optimization and correctness guarantees don't hinge on it since the VMM frequently gets control anyway and always checks for completions.

Figure 2 shows the additional data flow and computation in the system to accomplish our goal of reducing IPIs. The primary concern is that a guest OS might have scheduled a compute intensive task, which may result in the VMM not receiving an intercept. In the worst case, the VMM will wait until the next timer interrupt, which could be several milliseconds away, to receive a chance to execute and deliver virtual interrupts. So, our goal is to avoid delivering IPIs as much as possible while also bounding the extra latency increase.

We introduce as part of the shared area between the VMM and the VMkernel where completion queues are managed, a new time-stamp of the last time the VMM posted an IO completion virtual interrupt to the guest (see last line of Algorithm 2). We added a new step (3.5) in the VMkernel where before firing an IPI, we check the current time against what the VMM has posted to the shared area. If the time difference is greater than a configurable threshold, we post the IPI. Otherwise, we give the VMM an opportunity to notice IO completions in due course on its own. Section 4.5 provides experimental evaluation of the impact of IPI delay threshold values.

3.4 Implementation Cost

We took great care to minimize the cost of vIC and to make our design and implementation as portable as possible. A part of that was to refrain from using any floating point code. In the critical path code (Algorithm 2), we even avoid integer divisions. This should allow our design to be directly implementable in other hypervisors on any CPU architecture, and even in firmware or hardware of storage controllers. For reference, the increase in the 64-bit VMM `.text` section was only 400 bytes and the `.data` section grew by only 104 bytes. Our patch for the LSI Logic emulation in the VMM was less than 120 LoC. Similarly, the IPI coalescing logic in the VMkernel was implemented with just 50 LoC.

4 Experimental Evaluation

To evaluate our vIC approach, we have examined several micro-benchmark and macro-benchmark workloads and compared each workload with and without interrupt coalescing. In each case we have seen a reduction in CPU overhead, often associated with an increase in throughput (IOPS). For all of the experiments, unless otherwise indicated, the parameters are set as follows: $cifThreshold = 4$, $iopsThreshold = 2000$ and $epochPeriod = 200\,ms$. All but the TPC-C experiments were run on an HP Proliant DL-380 machine with 4 dual-core AMD 2.4 GHz processors. The attached storage array was an EMC CLARiiON CX3-40 with very small fully cached LUNs. The Fibre Channel HBA used was a dual-port QLogic 4Gb card.

In the next subsections, we first discuss the results for the Iometer micro benchmark in Section 4.1. Next, we cover the CPU utilization improvements of the Iometer benchmark and a detailed break-down of savings in Section 4.2. Section 4.3 presents our evaluation of vIC using two SQL IO simulators, namely SQLIOSim and GSBlaster. Finally, we present results for a complex TPC-C-like workload in Section 4.4. For each of the experiments, we have looked at the CPU savings along with the impact on throughput and latency of the workload.

4.1 Iometer Workload

We evaluated two Iometer [1] workloads running on a Microsoft Windows 2003 VM on top of an internal build of VMware ESX Server. The first workload consists of 4KB sequential IO reads issued by one worker thread running on a fully cached Logical Unit (LUN). In other words, all IO requests are hitting the array's cache instead of requiring disk access. The second workload is identical except for a different block size of 8KB.

For both workloads we varied the number of outstanding IOs to see the improvement over baseline. In Table 2, we show the full matrix of our test results for the 4KB workload. Furthermore, Table 3 summarizes the percentage improvements over the baseline where coalescing was disabled. The column labeled *R*, in Table 2, is the average ratio chosen by algorithm based on varying CIF over the course of the experiment; as expected, our algorithm coalesces more rigorously as the number of outstanding IOs is increased. Looking closely at the 64 CIF case, we can see that the dynamic delivery ratio, *R*, was found to be 1/6 on average. This means that one interrupt was delivered for every six IOs. The guest operating system reported a drop from 113K interrupts per

OIO	$\hat{R}$	IOPS	CPU cost cycles/ IO	Int/sec Guest	Baseline IOPS	Baseline CPU Cost	Baseline Int/sec Guest
8	4/5	31.2*K*	82.6*K*	49*K*	30.5*K*	84.2*K*	47*K*
16	2/3	38.9*K*	74.8*K*	58*K*	38.4*K*	77.0*K*	60*K*
32	1/3	48.3*K*	68.0*K*	69*K*	46.4*K*	70.5*K*	74*K*
64	1/6	53.1*K*	64.0*K*	34*K*	52.9*K*	78.4*K*	113*K*

Table 2: Iometer 4KB reads with one worker thread and a cached Logical Unit (LUN). $\hat{R}$ is the average delivery ratio set dynamically by the algorithm in this experiment. OIO is the number of outstanding IOs setting in Iometer. At runtime, CIF is often lower than the workload configured OIO as confirmed by $\hat{R}$ here being lower than the $R(OIO)$ from Table 1.

OIO	IOPS %diff	CPU cost %diff	Int/sec Guest %diff
8	2.3%	-1.9%	4.3%
16	1.3%	-2.8%	-3.3%
32	4.1%	-3.5%	-6.8%
64	0.4%	-18.4%	-66.4%

Table 3: Summary of improvements in key metrics with vIC. The experiments is the same as in Table 2.

second to 34K. The result of this is that the CPU cycles per IO have also dropped from 78.4K to 64.0K, which is an efficiency gain of 18%.

In Tables 4 and 5 we show the same results as before, but now with the 8KB IO workload. For the 64 CIF case, the algorithm results in the same interrupt coalescing ratio of 1/6 with now a 7% efficiency improvement over the baseline. The interrupt per second in the guest have dropped from 30K to 11K.

In both Table 2 and 4 we see a noticeable reduction in CPU cycles per IO whenever vIC has been enabled. We also would like to note that throughput never decreased and in many cases actually increased significantly.

4.2 Iometer CPU Usage Breakdown

For the 8KB sequential read Iometer workload with 64 outstanding IOs, we examined the breakdown between the VMM and guest OS CPU usage. Table 6 shows the monitor's abridged kstats. The `VMK_VCPU_HALT` statistic is the percent of time that the guest was idle. Notice that the guest idle time has increased which implies that the guest OS spent less time processing IO for the same effective throughput. The guest kernel runtime is measured by the amount of time we spent in the `TC64_IDENT`. Here we see a noticeable decrease in kernel mode execution time from 9.0% to 7.4%. The LSI Logic virtual SCSI adapter IO issuing time measured by `device_Priv_Lsilogic_IO` has decreased from

OIO	$\hat{R}$	IOPS	CPU cost cycles/ IO	Int/sec Guest	Baseline IOPS	Baseline CPU Cost	Baseline Int/sec Guest
8	4/5	31.2*K*	83.6*K*	48*K*	29.9*K*	88.2*K*	49*K*
16	2/3	39.3*K*	77.6*K*	61*K*	38.5*K*	81.3*K*	63*K*
32	1/3	41.5*K*	76.0*K*	60*K*	41.1*K*	77.1*K*	69*K*
64	1/6	41.5*K*	71.0*K*	11*K*	41.1*K*	75.7*K*	30*K*

Table 4: Iometer 8KB reads with one worker thread and a cached Logical Unit (LUN). Caption as in Table 2.

OIO	IOPS %diff	CPU cost %diff	Int/sec Guest %diff
8	4.3%	-5.2%	-2.0%
16	2.1%	-4.5%	-3.2%
32	1.0%	-1.5%	-13.0%
64	1.0%	-6.2%	-63.3%

Table 5: Summary of improvements in key metrics with vIC. The experiments is the same as in Table 4.

5.0% to 4.3%.

The IO completion work done in the VMM is part of a generic message delivery handler function and is measured by `DrainMonitorActions` in the profile. The table shows a slight increase from 0.5% to 0.7% of CPU consumption due to the management of the interrupt coalescing ratio.

The net savings gained by enabling virtual interrupt coalescing can be measured by looking at the guest idle time which is a significant 6.4% of a core. In a real workload which performs both IO and CPU-bound operations, this would result in an extra 6+% of available time for computation. We expect that some part of this gain also includes the reduction of the virtualization overhead as a result of vIC mostly consisting of second order effects related to virtual device emulation.

4.3 SQLIOSim and GSBlaster

We also examined the results from SQLIOSim [13] and GSBlaster. Both of these macro-benchmark workloads are designed to mimic the IO behavior of Microsoft SQL Server.

SQLIOSim is designed to target an "ideal" IO latency to tune for. That means that if the benchmark sees a higher IO latency it assumes that there are too many outstanding IOs and reduces that number. The reverse case is also true allowing the benchmark to tune for this preset optimal latency value. The user chooses this value to maximize their throughput and minimize their latency. In SQLIOSim we used the default value of 100ms.

GSBlaster is our own internal performance testing tool which behaves similar to SQLIOSim. It was designed as a simpler alternative to SQLIOSim which we could

	With vIC	Without vIC
VMK_VCPU_HALT	71.4%	65.0%
TC64_IDENT	7.4%	9.0%
device_Priv_Lsilogic_IO	4.3%	5.0%
DrainMonitorActions	0.7%	0.5%

Table 6: VMM profile for 8KB sequential read Iometer workload. Each row represents time spent in the related activity relative to a single core. The list is filtered down for space reasons to only the profile entries that changed.

	IOPS	CPU Cost	Baseline IOPS	Baseline CPU Cost	IOPS %diff	CPU Cost %diff
SQLIOSim	6282	339*K*	5327	410*K*	+17.9%	-17.4%
GSBlaster	24651	126*K*	20755	151*K*	+18.8%	-16.6%

Table 7: Performance improvements in SQLIOSim and GSBlaster. Improvements are seen both in IOPS and CPU efficiency.

understand and analyze in an easier manner. As opposed to SQLIOSim, when using GSBlaster we choose a fixed value for the number of outstanding IOs. It will then run the workload based on this configuration.

Table 7 shows the results of our optimization on both the target macro-benchmark workloads. We can see that the IOPS increased as a result of vIC by more than 17%. As in previous benchmarks, we also found that the CPU cost per IO decreased (by 17.4% in the case of SQLIOSim and 16.6% in the case of GSBlaster).

4.4 TPC-C Workload

The results seen so far have been for micro and macro benchmarks with relatively simple workload drivers. Whereas such data gives us insight into the upside potential of vIC, we have to put that in context of a large, complex application that performs computation as well as IO. We chose to evaluate our system on our internal TPC-C testbed[1]. The number of users is always kept large enough to fully utilize the CPU resources, such that adding more users won't increase the performance. Our TPC-C run was on a 4-way Opteron E based machine. The system was backed by a 45-disk EMC CX-3-20 storage array.

Table 8 shows results with and without interrupt coalescing with a $cifThreshold$ range of 2–4. When vIC is enabled, we were able to increase the number of users from 80 to 90 at the CPU saturation point. This immediately demonstrates that vIC freed up more CPU for real workload computation. Increasing the number of users also helps increase the achieved, user-visible benchmark

[1]Non-comparable implementation; results not TPC-C™compliant; deviations include: batch benchmark, undersized database.

	T	T Diff	Users	IOPS	Intr/ Sec	Latency
No vIC	43.3		80	10.2K	9.9K	7.7ms
$cifT = 4$	44.6	+3.0%	90	10.4K	6.4K	8.5ms
$cifT = 2$	45.5	+5.1%	90	10.5K	5.8K	9.2ms

Table 8: TPC-C workload throughput (T) run with and without interrupt coalescing and with different cifThreshold (cifT) configuration parameter values.

	Diff vs. baseline	
	$cifT = 4$	$cifT = 2$
IDT_AfterInterrupt	-28%	-31%
device_Priv_Lsilogic_IO	-12%	-14%
DrainMonitorActions	-19%	-22%
Intr_Deliver	-25%	-30%
Intr_IOAPIC	-40%	-46%
device_SCSI_CmdComplete	-13%	-16%
Intr	-42%	-49%

Table 9: VMM profile for TPC-C. Each row represents improvements in time spent in the related activity. The list is filtered down for space reasons to only the profile entries that changed significantly.

metric of throughput or transactions per minute. Table 8 shows that the transcation rate increased by 3.0% and 5.1% for $cifThreshold$ of 4 and 2, respectively. In our experience, optimizations for TPC-C are very difficult and significant investment is made for each fractional percent of improvement. As such, we consider an increase of 3.0%–5.1% to be very significant.

We also logged data for the algorithm-selected vIC rate, R, every 200 ms during the run. Figure 3 shows the histogram of the distribution of R for certain duration of the TPC-C run. It is interesting that R varies dramatically throughout the workload. The frequency distribution here is a function of the workload and not vIC.

Furthermore, in Figure 4 we show the same data plotted against time. The interrupt coalescing rate varies significantly because the TPC-C workload has periods of IO bursts. In fact, there are numerous times where the algorithm selects to deliver less than 1 out of every 12 interrupts. This illustrates how the self-adaptability and responsiveness of our algorithm is necessary to satisfy complex real-world workloads.

As a result of interrupt coalescing, we see a decrease in virtual interrupts per second delivered to the guest from 9.9K to 6.4K and 5.8K. We also noticed an increase in the IOPS achieved by the storage array. This can be explained by the fact that there is increased parallelism (more users) in the input workload. Such a change in workload has been demonstrated in earlier work to increase throughput [11].

Any increase in parallelism is also accompanied by an expected increase in average latency [11]. This explains

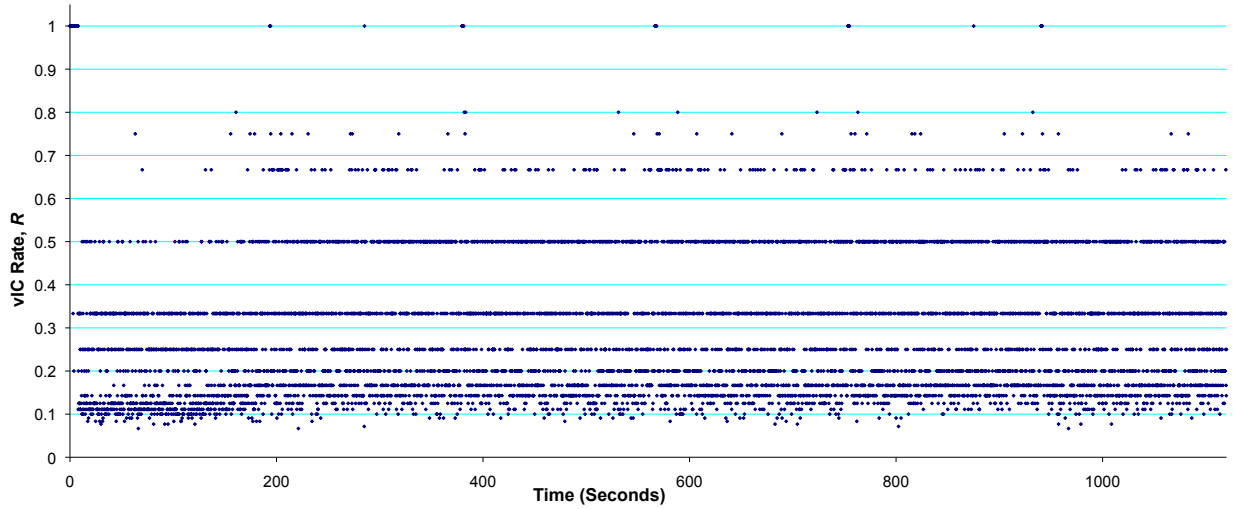


Figure 4: The algorithm selected virtual interrupt coalescing rate, R, over time for TPC-C. The high dynamic range illustrates the burstiness in outstanding IOs of the workload and the resulting online adaptation by vIC.

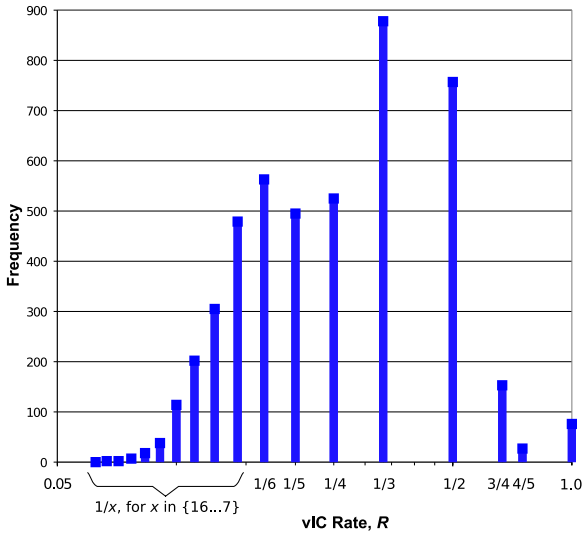


Figure 3: Histogram of dynamically selected virtual interrupt coalescing rates, R, during our TPC-C run. The x-axis is log-scale.

the bulk of the increase between the no-vIC and the vIC-enabled rows in Table 8. However, one would expect a certain increase in latency from any interrupt coalescing algorithm. In our case, we expect latency increases to be less than a few hundred microseconds. Using instantaneous IOPS, CIF and R, we can calculate the mean delay from vIC. For instance, at the median delivery rate $R = 1/3$, at the achieved mean IOPS of 10K for this experiment, the increase would be 200 μs.

In Table 9 we show a profile of percentage reduction in CPU utilization of several key VMM components as a result of interrupt coalescing. IOs are processed as part of the VMM's action processing queue. The reduction in the queue processing is between 19% and 22% for the CIF thresholds of 4 and 2 respectively. Fewer interrupts means that the guest operating system is performing fewer register operations on the virtual LSI Logic controller, shown by `device_Priv_Lsilogic_IO`. The net reduction in device operations translated to a 12% and 14% reduction, respectively, in CPU usage relative to using the virtual device without vIC. We also measured an approximately 30% reduction in the monitor's interrupt delivery function `Intr_Deliver`.

4.5 IPI interference: CPU-bound loads

Recall that we described an optimization of not posting IPIs in all cases using a threshold delay, in order to lower the impact of IPIs on VMs workload (Section 3.3). In this section, we first motivate our optimization by showing that the impact on CPU-bound applications can be very high. We show nevertheless that sending at least some IPIs is essential as an IO latency and throughput optimization. We then provide data to illustrate the difficult trade-off between the two concerns.

We ran two workloads, one IO bound and the other CPU bound, on the *same* virtual machine running the Microsoft Windows 2003 operating system. The IO-bound workload is running an Iometer, 1 worker benchmark, doing 8 OIO, 8K Read from a fully cached small LUN. The CPU-bound workload is an SMP version of Hyper Pi for which the run time in seconds to calculate 1M digits of π on two virtual processors is plotted as triangles (average of 6 runs). Hyper Pi is fully-backlogged and we run

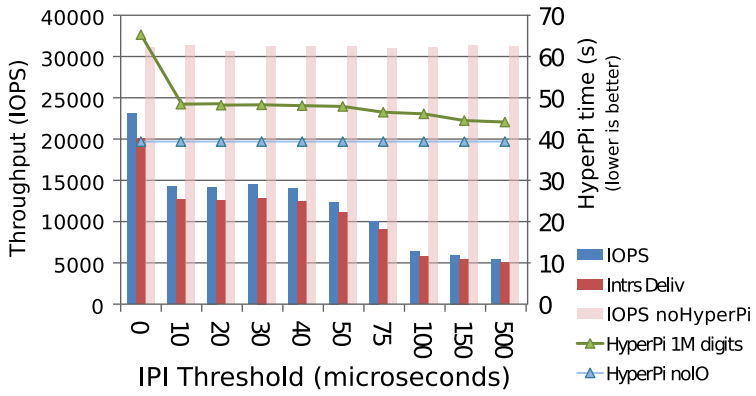


Figure 5: Effect of different IPI send thresholds. This plot illustrates the trade-off in IO throughput and CPU efficiency of two co-running benchmarks, as we vary the key parameter: the IPI send threshold. The workloads are run on the same 2-vCPU Windows 2003 guest. The IO workload performance is shown as vertical bars for an Iometer, 1 worker, 8 OIO, 8K Read from a fully cached small LUN. Each IO data point is the average of 50 consecutive samples. The CPU-bound workload is an SMP version of Hyper Pi for which the run time in seconds to calculate 1M digits of *pi* on two virtual processors is plotted as triangles (average of 6 runs). For each workload, both the co-running and independent scores are plotted. No delay in sending an IPI results in the highest IO throughput whereas waiting 500 microseconds to send an IPI results in the highest performance of the CPU-bound workload.

it at the `idle` priority class effectively giving Iometer higher priority while ensuring that the guest never halts due to idling.

Figure 5 shows the effect of different IPI send thresholds on performance. This plot illustrates the trade-off in IO throughput and CPU efficiency of two co-running benchmarks respectively, as we vary the IPI send threshold from VMkernel. For each workload, Figure 5 shows both the co-running and independent scores. The IO workload performance is shown as vertical bars in terms of IO throughput observed by Iometer. Each IO data point is the average of 50 consecutive samples. Performance of Hyper Pi workload is shown in terms of time to completion of 1 million digit computation for π.

First, notice that the throughput for "IOPS no Hyper Pi" bars do not change much with the IPI send threshold. This is because the guest is largely idle and frequently entering the CPU halt state. Whenever the guest halts, the VMM gains execution control and has the chance to check for pending completions from the VMkernel. As such, sending IPIs more or less frequently has hardly any bearing on the latency of the VMM noticing and delivering an interrupt to the guest. Similarly, the blue triangle case of "Hyper Pi no IO" shows no sensitivity to our parameter. This is obvious: no IO activity means that IPIs are out of the picture anyway.

As soon as we run those two benchmarks together, they severely interfere with each other. This is due to the fact that Hyper Pi is keeping the CPU busy implying that the guest never enters halt. Therefore, the VMM only has rare opportunities to check for pending IO completions (*e.g.*, when a timer interrupt is delivered). Hyper Pi sees its best performance at high IPI-send thresholds (right end of the x-axis) since it gets to run uninterrupted for longer periods of time. However, IO throughput (see the blue bars) suffers quite a bit from the increased latency. Conversely, if no delay is used (left end of the x-axis), the impact on the performance of Hyper Pi is severe. As expected, IO performance does really well in such situations.

It should be noted that the two workloads, though running on the same virtual machine, are not tied to each other. In other words, there is no closed loop between them as is often the case with enterprise applications. Still, the setup is illustrative in showing the impact of IPIs on IO and CPU bound workloads.

These results indicate that setting a good IPI send threshold is important but nontrivial and even workload dependent. We consider automatic setting of this parameter to be highly interesting future work. For ESX, we have currently set the default to be 100 microseconds to favor CPU bound parts of workloads based on experiments on closed loop workloads run against real disks.

5 Deployment Experience

Our implementation of vIC as described in this paper has been the default for VMware's LSI Logic virtual adapter in the ESX hypervisor since version 4.0, which was released in the second quarter of 2009. As such, we have significant field experience with deployments into the tens of thousands of virtual machines. Since then, we

have not received any performance bug reports on the virtual interrupt coalescing algorithm.

On the other hand, the `pvscsi` virtual adapter which first shipped in ESX 4.0 did not initially have some of the optimizations that we developed for the LSI Logic virtual interrupt coalescing. In particular, although it had variable interrupt coalescing rates depending on CIF, it was missing the *iopsThreshold* and was not capable of setting the coalescing rate, R, between $1/2$ and 1. As a result, several performance bugs were reported. We triaged these bugs to be related to the missing optimizations in `pvscsi` which are now fixed in the subsequent ESX 4.1 release. We feel that this experience further validates the completeness of our virtual interrupt coalescing approach as a successful, practical technique for significantly improving performance in one dimension (lower CPU cost of IO) without sacrificing others (throughput or latency).

The key issue with any interrupt coalescing scheme is the potential for increases in IO response times which we have studied above. At high IO rates, some application *IO* threads might be blocked a little longer due to coalescing. In our case, this delay is strictly bound by the $1/iopsThreshold$. Our solution is significantly better than other coalescing techniques since it explicitly takes CIF into account. In our experience, vIC lets *compute* threads of real applications run longer before getting interrupted. Increased execution time can reduce overhead from sources such as having the application's working set evicted from the CPU caches, etc. Interrupt coalescing is all about the trade off between CPU efficiency and IO latencies. Hence, we provide parameters to adjust that tradeoff if necessary, though the default settings have been tuned using a variety of workloads.

6 Related Work

Interrupts have been in active use since early days of computers to handle input-output devices. Smotherman [19] provides an interesting history of the evolution of interrupts and their usage in various computer systems starting from UNIVAC (1951). With increasing network bandwidth and IO throughput for storage devices, the rate of interrupts and thus CPU overhead to handle them has been increasing pretty much since the interrupt model was first developed. Although processor speeds and number of cores have been increasing to keep up with these devices, the motivation to reduce overall CPU overhead of interrupt handling has remained strong. Interrupt coalescing has been very successfully deployed in various hardware controllers to mitigate the CPU overhead. Many patents and papers have been published on performing interrupt coalescing for network and storage hardware controllers.

Gian-Paolo's patent [15] provides a method for dynamic adjustment of maximum frame count and maximum wait time parameters for sending the interrupts from a communication interface to a host processor. The packet count parameter is increased when the rate of arrivals is high and decreased when the interrupt arrival rate gets low. The maximum wait time parameter ensures a bounded delay on the latency of the packet delivery. Hickerson and Mccombs's patent [12] uses a single counter to keep track of the number of initiated tasks. The counter is decremented on the task completion event and it is incremented when the task is initiated. A delay timer is set using the counter value. An interrupt is generated either when the delay timer is fired or the counter value is less than a certain threshold. In contrast to both of these patented techniques, our mechanism adjusts the delivery rate itself based on CIF and does not rely on any delay timers. It should be noted, however, that our approach is complementary to interrupt coalescing optimizations done in the hardware controllers since they can benefit in lowering the load on the hypervisor host, in our case the ESX VMkernel.

QLogic [4] and Emulex [5] have also implemented interrupt coalescing in their storage HBAs but the details of their implementation are not publicly available. The knowledge base article for a Qlogic driver [3] suggests the use of a delay parameter, *ql2xintrdelaytimer* which is used as a wait time for firmware before generating an interrupt. This is again dependent on high resolution timers and delaying the interrupts by a certain amount. Online documentation suggests that the QLogic interrupt delay timer can be set in increments of 100 μs. Interrupt coalescing can be disabled by another parameter called *ql2xoperationmode*. Interestingly, this driver parameter allows two modes of interrupt coalescing distinguished by whether an interrupt is fired if CIF drops to 0. A similar document related to Emulex device driver for VMware [2] suggests the use of statically defined delay and IO count thresholds, *lpfc_cr_delay, lpfc_cr_count*, for interrupt coalescing.

Stodolsky, Chen and Bershad [20] describe an optimistic scheme to reduce the cost of interrupt masking by deferring the processing ("continuation") of any interrupt that arrives during a critical section to a later time. Many operating systems now use similar techniques to handle the scheduling of deferred processing for interrupts. The paper also suggests that interrupts be masked at the time of deferral so that the critical section can continue without further interruptions. Level-triggered interrupts like the ones described in our work are another way of accomplishing the same thing. Both of these techniques from the Bershad paper are complementary to the idea of coalescing which is more concerned with the *delay* of interrupt delivery.

Zec et al. [22] study the impact of generic interrupt coalescing implementation in 4.4BSD on the steady state TCP throughput. They modified the *fxp* driver in FreeBSD and controlled only the delay parameter T_d, which specifies the time duration between the arrival of first packet and the time at which hardware interrupt is sent to the OS. Similarly, Dong et al. recently studied [8] the CPU impact of interrupts and proposed an adaptive interrupt coalescing scheme for a network controller.

Mogul and Ramakrishnan [14] studied the problem of receive livelock, where the system is busy processing interrupts all the time and other necessary tasks are starved to the most part. To avoid this problem they suggested the hybrid mechanism of polling under high load and using regular interrupts for lighter loads. Polling can increase the latency for IO completions, thereby affecting the overall application behavior. They optimized their system by using various techniques to initiate polling and enable interrupts under specific conditions. They also proposed round robin polling to fairly allocate resources among various sources.

Salah et al. [17] did an analysis of various interrupt handling schemes such as polling, regular interrupts, interrupt coalescing, and disabling and enabling of interrupts. Their study concludes that no single scheme is good under all traffic conditions. This further motivates the need for an adaptive mechanism that can adjust to the current interrupt arrival rate and other workload parameters. Salah [16] performed an analytical and simulation study of the relative benefit of time-based versus number-of-packets based interrupt coalescing in context of networking. More recently Salah and Qahtan [18] implemented and evaluated a different hybrid interrupt handling scheme for Gigabit NICs in Linux kernel 2.6.15. Their hybrid scheme switches between interrupt disabling-enabling (DE) and polling.

Our approach, instead of switching to polling, adjusts the overall interrupt delivery rate during high load. We believe this is more flexible and adapts well to drastic changes in guest workload. We also use CIF which is available only in context of storage controllers but allows us to solve this problem more efficiently. Furthermore, we do not have the luxury to change the guest behavior in terms of interrupts vs polling because the guest OS is like a black box to virtualization hypervisors.

7 Conclusions

In this paper, we studied the problem of efficient virtual interrupt coalescing in context of virtual hardware controllers implemented by a hypervisor. We proposed the novel techniques of using the number of commands in flight to dynamically adjust the interrupt delivery ratio in fine-grained steps and to use future IO events to avoid the need of high-resolution timers. We also designed a technique to reduce the number of inter-processor interrupts while keeping the latency bounded. Our prototype implementation in the VMware ESX hypervisor showed that we are able to improve application throughput (IOPS) by up to 19% and improve CPU efficiency up to 17% (for the GSBlaster and SQLIOSim workloads respectively). When tested against our TPC-C workload, vIC improved the workload performance by 5.1% and demonstrated the ability of our algorithm to adapt quickly to changes in the workload. Our technique is equally applicable to hypervisors and hardware storage controllers; we hope that our work spurs further work in this area.

8 Open Problems

There are some open problems which deserve further exploration by our fellow researchers and practitioners. Firmware implementations of vIC could lower the cost of hardware controllers and provide tighter latency control than what is available today. Currently, our vIC implementation hard-codes the best CIF-to-*R* mappings based on extensive experimentation. Dynamic adaptation of that mapping appears to be an interesting problem. In some architectures, PCI devices are directly passed-through to VMs. Interrupt coalescing in this context is worthy of investigation.

At first blush, networking controllers do not appear to lend themselves to a CIF-based approach since the protocol layering in the stack means that the lower layers (where interrupt posting decisions are made) do not know the semantics of higher layers. Still, we speculate that inference techniques might be applicable to do aggressive coalescing without loss of throughput in context of high-bandwidth TCP connections using window size-based techniques.

Acknowledgements

We would like to thank Maxime Austruy who worked on the `pvscsi` virtual adapter and co-invented some of the interrupt coalescing techniques discussed here. Many thanks to Davide Bergamasco, Jinpyo Kim, Vincent Lin and Reza Taheri for help with experimental validation of our work and to our shepherd, Muli Ben-Yehuda, for detailed comments and guidance. Finally, this work would not have been possible without support, encouragement and feedback from Ole Agesen, Mateen Ahmad, Keerti Garg, Mallik Mahalingham, Tim Mann, Glen McCready and Carl Waldspurger.

References

[1] Iometer. http://www.iometer.org.

[2] *Emulex Driver for VMware ESX*. August 2007. www-dl.emulex.com/support/vmware/732/vmware.pdf.

[3] *QLogic: Advanced Parameters For Driver Modules Under VMware*. October 2009. http://kb.qlogic.com/KanisaPlatform/Publishing/548/1492_f.SAL_Public.html.

[4] *QLogic: QLE8152 datasheet*. 2009. http://www.starline.de/fileadmin/images/produkte/qlogic/QLogic_QLE8152.pdf.

[5] *Emulex: OneCommand Manager*. June 2010. http://www.emulex.com/artifacts/ad19cc4e-870a-42e9-a4b2-bcaa70e2afd6/elx_rc_all_onecommand_efficiency_qlogic.pdf.

[6] I. Ahmad. Easy and Efficient Disk I/O Workload Characterization in VMware ESX Server. *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, pages 149–158, Sept. 2007.

[7] X. Chang, J. Muppala, Z. Han, and J. Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *IEEE International Conference on Communications(ICC)*, pages 1835–1839, May 2008.

[8] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H. Guan. High performance network virtualization with SR-IOV. In *HPCA*, pages 1–10, 2010.

[9] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportionate Allocation of Resources for Distributed Storage Access. In *Proc. Conference on File and Storage Technology (FAST '09)*, Feb. 2009.

[10] A. Gulati, C. Kumar, and I. Ahmad. Storage Workload Characterization and Consolidation in Virtualized Environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[11] A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing across Storage Devices. In *Proc. Conference on File and Storage Technology (FAST '10)*, Feb. 2010.

[12] R. Hickerson and C. C. Mccombs. Method and apparatus for coalescing i/o interrupts that efficiently balances performance and latency. (US PTO 6065089), May 2000.

[13] Microsoft. How to use the sqliosim utility to simulate sql server activity on a disk subsystem, 2009. http://support.microsoft.com/kb/231619.

[14] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[15] G. paolo D. Musumeci. System and method for dynamically tuning interrupt coalescing parameters. (US PTO 6889277), May 2005.

[16] K. Salah. To coalesce or not to coalesce. *Intl. J. of Elec. and Comm.*, pages 215–225, 2007.

[17] K. Salah, K. El-Badawi, and F. Haidari. Performance analysis and comparison of interrupt-handling schemes in gigabit networks. *Comput. Commun.*, 30(17):3425–3441, 2007.

[18] K. Salah and A. Qahtan. Implementation and experimental performance evaluation of a hybrid interrupt-handling scheme. *Comput. Commun.*, 32(1):179–188, 2009.

[19] M. Smotherman. Interrupts, 2008. http://www.cs.clemson.edu/~mark/interrupts.html.

[20] D. Stodolsky, J. B. Chen, and B. N. Bershad. Fast interrupt priority management in operating system kernels. In *moas'93: USENIX Symposium on USENIX Microkernels and Other Kernel Architectures Symposium*, pages 9–9, Berkeley, CA, USA, 1993. USENIX Association.

[21] VMware, Inc. *Introduction to VMware Infrastructure*. 2010. http://www.vmware.com/support/pubs/.

[22] M. Zec, M. Mikuc, and M. Zagar. Estimating the impact of interrupt coalescing delays on steady state tcp throughput. *Tenth SoftCOM 2002 conference*, 2002.

[23] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS*, Sep 2005.

Power Budgeting for Virtualized Data Centers

Harold Lim
Duke University

Aman Kansal
Microsoft Research

Jie Liu
Microsoft Research

Abstract

Power costs are very significant for data centers. To maximally utilize the provisioned power capacity, data centers often employ over-subscription, that is, the sum of peak consumptions of individual servers may be greater than the provisioned capacity. Power budgeting methods are employed to ensure that actual consumption never exceeds capacity. However, current power budgeting methods enforce capacity limits in hardware and are not well suited for virtualized servers because the hardware is shared among multiple applications. We present a power budgeting system for virtualized infrastructures that enforces power limits on individual distributed applications. Our system enables multiple applications to share the same servers but operate with their individual quality of service guarantees. It responds to workload and power availability changes, by dynamically allocating appropriate amount of power to different applications and tiers within applications. The design is mindful of practical constraints such the data center's limited visibility into hosted application performance. We evaluate the system using workloads derived from real world data center traces.

1 Introduction

Data centers require large amounts of power and the costs of their power supply infrastructure, backup generators and batteries, and power consumption are a significant concern [13]. Aside from costs, the availability of power may be a limiting factor, especially for smaller data centers deployed in enterprise buildings, educational institutions, and for emerging container-based "edge" data centers located close to end users. As a result, data center design must minimize the power capacity requested from utilities. The need to optimize provisioned power capacity has lead to the adoption of a practice known as *over-subscription*. In over-subscribed data centers, the sum of the possible peak power consumptions of all the servers combined is greater than the provisioned capacity. Servers typically operate below their peak power and even when servers from one application are near peak usage, other servers may be well below their peaks, keeping the total power within capacity. To ensure that actual total power use stays below capacity, servers are equipped with power budgeting mechanisms that can throttle the power usage of a server, such as by reducing the processor frequency. Power budgeting has been used for several safe and efficient over-subscription methods [9, 3, 23, 19].

However, the current methods are not well suited to virtualized infrastructures where the servers are shared by virtual machines (VMs) belonging to different applications, due to several reasons. First, in virtualized infrastructures, there is a disconnect between the *physical* server layout and the *logical* organization of resources among applications. Hardware power budgeting used in current power budgeting methods does not respect the isolation among virtual machines with different performance requirements. Second, existing techniques do not explicitly address workload and power dynamics. As input workload volumes change, the power available for different applications changes, as does the optimal distribution of power among an application's constituent tiers. Third, existing designs typically use a single power control knob and do not exploit multiple feasible combinations of power settings for optimizing performance.

In this paper we present a power budgeting solution named virtualized power shifting (VPS) that efficiently coordinates the power distribution among a large number of VMs within given peak power capacity. VPS dynamically shifts power among various distributed components to efficiently utilize the total available power budget, as workloads and power availability vary. Power is distributed among application components in the correct proportions to achieve the best performance. The system respects application boundaries and differentiates

performance based on priorities. In contrast to existing techniques that use only one power control knob, typically frequency scaling, VPS uses multiple power control knobs and selects the optimal combinations of power settings to optimize performance within the available power budget. We describe how the system operates with practical constraints such as limited insight into application performance.

The system tracks dynamic power availability and workload dynamics with low error, as its design is based on well-studied control theoretic algorithms with desirable stability and accuracy properties. We evaluate the system through experiments on a multi-server testbed running a mix of interactive and batch processing benchmarks. Real world data center traces from Microsoft's online services are used.

2 Virtualized Power Budgeting Challenges

The over-arching problem addressed by VPS is to dynamically adjust power allocations in a multi-application scenario. The key challenges presented by this problem are discussed below.

2.1 Server Sharing

The large number of servers in a data center are shared among multiple applications, typically using virtualization. VMs from multiple applications may be co-located on the same physical servers depending on the application characteristics such as complimentary resource usage and data placement needs. At the same time, VMs from one application are spread across many servers based on required minimum spread for hardware redundancy, and minimum number of servers needed to allow for seamless software upgrades. Since different applications have different users and workloads, an increase in the workload and power usage of one application should not negatively impact another application sharing the same hardware. In some scenarios, different applications may have different priorities. For example, customer-facing online services may have higher priority than batch processing or internal enterprise applications.

The power budgeting mechanism must therefore enforce power limits at *application granularity* rather than at the hardware level. Power budgets enforced in hardware, such as using dynamic voltage and frequency scaling (DVFS), impact an entire server or all processor cores supplied from the same voltage rail[1]. This will cause a performance drop for all application VMs sharing those processors. Additional power capping mechanisms that operate at the individual VM level must be used. Coordination of power allocations must also follow the application VM layout across the server infrastructure.

[1] In most servers, an entire processor chip or socket is supplied from a single voltage rail and hence the supply voltage and DVFS can only be controlled for the entire socket consisting of multiple cores.

2.2 Multi-dimensional Power Control

To respect application boundaries, a combination of hardware-based (e.g. DVFS) and software-based (e.g. VM CPU time allocation) power control knobs is used. Multiple knobs imply that more than one combinations of power settings may achieve the same power level. However, application performance may be different for each feasible combination. We illustrate this phenomenon through experimental measurements in a later section (Figure 5): at a given power level, performance varies up to 25% depending on power settings. Power budgeting design has the opportunity to *maximize performance*, if it intelligently selects the best combination of power settings to satisfy the power budget. Optimization of performance brings with it challenges of measuring and modelling performance. These measurements may not be available in certain scenarios, especially when the applications are not owned by the same entity that manages the data center, as is often the case for large organizations and cloud based infrastructures. The VPS system includes different modes of operation to work with or without such information.

2.3 Dynamic Power Proportions

The input workload volume for each application changes over time, implying that the power used, and as a result the power available for other applications, changes. The power budgets must be dynamically adapted, requiring run time coordination across all applications.

Within an application, allocation of power to its VMs is also non-trivial since the best allocation may vary with workload volume. This happens because different VMs may be hosting different tiers of the application. As a toy example, consider a two tier application with the front-end tier executing a processor intensive stage and the back-end tier providing data storage. Power usage of the front-end tier depends on processor utilization, and as an example suppose it changes between 50W and 100W. The back-end comprises disk storage, and has a high idle power for keeping the disks spinning, say 80W, with an additional power usage of up to 20W that varies with the volume of I/O activity. At peak load, the allocation of power is 100W to each tier, while at idle, the power allocation is 50W and 80W to the two tiers respectively. The power distribution proportion among tiers is not constant.

Changes in workload not only change the application power consumptions but also influence the power

used by the shared components such as cooling equipment. Another dynamic factor is the power capacity available. For example, if environmentally-harvested energy is used, the available supply varies over time. In view of the workload and power capacity dynamics, to maximally utilize the available capacity, the budgeting mechanism must *dynamically adjust the power allocation proportions* across applications and application tiers.

3 System Design

The VPS system for power budgeting is designed to address the challenges described in the previous section. To support multiple distributed applications in a scalable manner, we use a hierarchical approach. The hierarchy is designed to follow the application layout, with the total power budget being divided dynamically among applications, and within the application among the different tiers, following down to the individual VMs comprising those tiers. This hierarchy is independent of the server and rack layout. Power is tracked at the VM level allowing each application to be budgeted independently of others by leveraging VM specific power control knobs (such as VM CPU time allocation). The dynamics of the system, including workload variations and power capacity changes, are handled through feedback controllers that monitor and control the power usage in real time. Application performance is optimized through a combination of control algorithms based on proportional-integral-derivative (PID) and model predictive control (MPC). In this section, we describe the design of the VPS architecture and the control algorithms used.

3.1 Power Budgeting Architecture

Figure 1 shows the overall structure of the VPS system. The white boxes correspond to VPS components while the gray boxes show the underlying physical hierarchy. The VPS control mechanism consists of a multi-level hierarchy where the topmost level is a *data center level controller*[2]. This controller receives the *total power capacity* as its input. The controller allocates it among the *application level controllers* that comprise the next level in the hierarchy. The top level controller is scalable since it monitors and controls only a small number of applications rather than the thousands of individual servers or VMs. Each application may in turn consist of hundreds or even thousands of VMs. The application is thus further divided into tiers and the application level controller monitors and controls only the *tier-level controllers*. The tiers of the application are typically arranged in a pipeline that, as we describe in the detailed controller design, facilitates our method for optimal power allocation among the application components. This approach utilizes the available power more efficiently than following a physical server or rack layout based hierarchy. The tier level controller in turn controls each VM belonging to the corresponding application tier. Within a tier, the constituent VMs are often load balanced and similar in behavior. If a particular tier has a reasonably small number of VMs, the tier level controller can perform a nearly uniform allocation within the tier, and directly command the VM power settings affecting the power consumption. If the number of VMs within a tier is very large, or the VM roles are distinct, further levels may be added to the hierarchy, based on network proximity or VM roles. Without loss of generality, we focus on a three-level hierarchy: data center level, application level, and tier level. VPS operates independent of the physical hierarchy comprising of servers, racks, and power distribution units (PDUs).

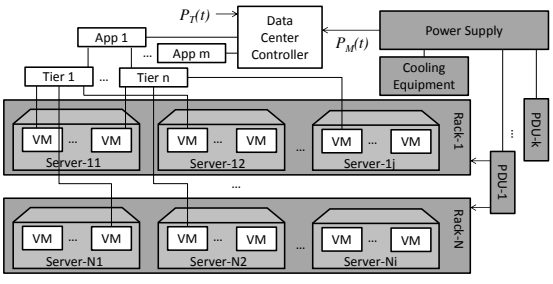


Figure 1: VPS power control hierarchy. The white boxes constitute VPS while the gray boxes represent the physical hierarchy. PDU stands for power Distribution unit.

The above architecture assumes that power can be measured and limits enforced at the application granularity. Measurement of an application's power consumption may not be possible at a physical wire in virtualized servers since the physical server components are shared across multiple applications. The VPS system relies on VM power measurement methods such as [8, 18] that report the individual power usage of each VM on a shared server, as well as, the base power that the hardware platform consumes. The individual VM power measurements are propagated up the VPS hierarchy to obtain the power consumption of each application or application tier.

Actual power reduction can only be realized at the lowest layer that controls the power consuming resource. In our implementation, the resource whose power is varied is the processor since in current platforms, the proces-

[2]A data center may be divided into sections referred to as "colos" where the power infrastructure and backup is separate for each colo: in this case one data center level controller would operate in each colo. The top level controller may be applied at any physical boundary representing an independent unit in terms of application deployment and power constraints.

sor is the resource with the most advanced power management options. By controlling the CPU active time allocated, i.e., the CPU utilization, the power consumption of the processor can be varied from near zero to its peak power. Similarly, DVFS also allows varying the processor power over a large range. We use both these knobs to control power. For memory, power varies directly with the number of IOs performed [8], which can effectively be throttled by the number of CPU cycles allocated. Additional power control knobs can be included in our framework as they become available since it is already designed to use multiple knobs. Applications whose power use does not change with any available power control knob can of course not be throttled. Their power use is measured, similar to idle power and cooling equipment power, and changes are compensated for by VPS controllers.

The application-based hierarchy is more natural from a performance perspective because resource allocation decisions are typically made for an application as a whole and each application has a different business functionality with its own priority, revenue, and QoS expectation. The implementation of such a hierarchy is however more sophisticated since it incorporates knowledge about the application's VM layout and the hypervisor's VM CPU time allocation. As a result, the VPS system operates in the data center management plane rather than in the server motherboard or blade-enclosure based firmware.

The power budget at the highest layer, denoted $P_T(t)$, is an input to VPS. This is the hard constraint that must be satisfied at all times. It could be based on the static capacity built for the facility, or could be dynamic, based on time-of-day based power prices, or amount of environmentally harvested energy [16, 2] available.

3.2 Top Level Controller

The data center level controller, placed the the top level in the design, determines the amount of power allocated to each application. If applications have different priorities, the controller takes those into account.

Naïvely partitioning the power budget $P_T(t)$ among applications, say based on statically assigned shares, does not work well in practice because of the following factors. First, the application workloads are dynamic. An application may not be using its fixed allocation at time t if incoming workload is low. The power allocation mechanism must adapt dynamically, to assign the unused power to another application if needed. Second, the measurement of application power consumption may have some errors. Additionally, a measurable power allocation increase to an application may lead to associated hidden power level increases in shared infrastructure, such as due to non-linear changes in transformation losses across power supplies and changes in cooling load, that are hard to assign to any single application. Such errors directly affect the total power used and must be compensated to satisfy the hard limit of $P_T(t)$.

VPS design uses feedback control to address the above factors. The top level controller receives measurements of each application's power consumption from the respective application level controllers. It also receives the total data center power consumption from hardware instrumentation at the power circuits supplying the servers and cooling equipment. This hardware measurement includes power consumption that is not directly attributed to any specific application VM. The output is the power allocation to each of the applications, at each time instance, that is then enforced by the application level controllers. Figure 2 shows the feedback loop involved.

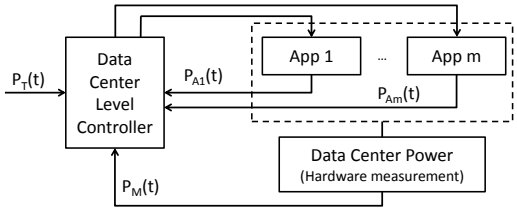


Figure 2: Block diagram of data center level feedback.

The only output action available at this controller is the power allocation and hence the performance objective is only to allocate the maximum possible power, up to the application demands, and minimize the workload throttling. Any controller that can closely track the available power limit and is robust to errors in measurements can be used, and a PID controller is thus appropriate at this layer. Other control algorithms that make optimal decisions by choosing among multiple control knobs are employed in VPS at lower layers. Ad hoc algorithms such as those based on rules that actuate power increases and decreases based on observed consumption levels may become unstable or oscillate as shown in [21]. VPS uses a control theoretic framework that enables stable operation by design, over the range of practical operating constraints (the specific methods used to tune the controllers in our prototype are outlined in Section 3.5).

3.2.1 Application Budgets and Priorities

Note that while the control algorithm adapts total power consumption to operate close to P_T, the PID controller output here is the sum of all the applications' power consumptions. There is no knob available to control this sum; only the power consumptions of individual applications can be affected through their respective application level controllers. The control output is split across the

application level controllers using application priorities as well as the measured application and total hardware power consumptions, as follows.

At power provisioning time, each application is assigned a maximum budget, based on expected usage at, say, 99-th percentile peak load. While in non-virtualized settings, application budgets are typically assigned simply based on number of servers allocated and measured power for those servers when running the relevant application, with virtualization the actual power impact must be profiled on the appropriate infrastructure. When a new application is accepted, its power can be profiled for each type of VM instance used, and extrapolated to number of VM instances at 99-th percentile of peak load. Such provisioning is typically required by a data center before it can accept an application to be hosted. Suppose the assigned power is denoted P^0_{Ai}.

Suppose $\Delta(t+1)$ denotes the desired change in total power at the next time step (PID controller output). To determine the per application power split, the amount of uncontrollable power, denoted $P_U(t)$, is first estimated by subtracting the sum of application power consumptions from the measured total power consumption, $P_M(t)$:

$$P_U(t) = P_M(t) - \sum_{i=1}^{m} P_{Ai}(t) \quad (1)$$

where $P_{Ai}(t)$ represents the power consumption of application i, and m is the number of applications. This uncontrolled portion of power includes all shared infrastructure power as well as errors in application power measurement. The estimate of the total power consumption at the next time step becomes $P_M(t) + \Delta(t+1)$. The power budget available to be allocated to the applications, denoted P_{app}, at the next time step, is estimated as:

$$P_{app}(t+1) = P_M(t) + \Delta(t+1) - P_U(t) \quad (2)$$

This is only an estimate since it does not include the unknown change in $P_U(t)$ at the next time step, and that change will act as an error for the feedback controller, to be compensated as the controller converges. $P_{app}(t+1)$ is distributed among the applications according to the desired priortization policy.

In our implementation, the priortization policy is as follows. The controller allocates power to each application based on its current demand, subject to a maximum of P^0_{Ai}, starting with the highest priority applications. If at any priority level, there is not enough power budget to satisfy all application power demands, then we use *weighted fair sharing* to distribute the remaining power, with weights set proportional to the initial provisioned application budgets. With this policy, lower priority applications are affected first. Similarly, any excess power left over is also assigned using weighted fair sharing to applications with unsatisfied demand (excess power may be available when some applications are below their initial budget). The assigned shares are sent to each application level controller to be enforced. Effectively, priority levels determine the split of P_{app} across applications while the feedback controller tunes the value of P_{app} to meet the target total power.

3.3 Application Level Controller

The VMs comprising an application are typically divided into a number of tiers. Each tier has a different role, and consequently a different power requirement. The application-level controller distributes the application power budget received from the top level controller among each of its application tiers. This controller only communicates with a small number of tier level controllers and is thus scalable in number of VMs.

The controller must determine the correct proportion in which power is allocated to the different tiers. One design option for this controller is to learn a model of power usage across tiers, and use that to determine the appropriate ratio in which power should be split among the tiers. This approach can be used when a detailed model of application performance and resource utilization at each tier can be learned. This is feasible for a specific power control knob at a given workload volume [7]. However, as illustrated in Section 2.3, the best power sharing proportion changes with workload volume. The model may also depend on the power control knob used at the lower layer, such as DVFS or CPU time allocation. Further, the application behavior may change over time with software upgrades. In a virtualized infrastructure supporting multiple applications, with little control over application internals, learning this model is difficult.

VPS design dynamically tunes the power allocations without relying on previously learned models. The key challenge of course is to determine the correct sharing ratio. Our design is based on the observation that the multiple application tiers are arranged in a pipeline, and throttling one tier will directly affect the workload flowing into other tiers. The relationship among power changes at different tiers need not be known a-priori, as long as the pipeline assumption holds. The VPS application-level controller measures the total application power usage but controls only one of the tiers. As the power allocation to the controlled tier is changed, the power consumed by other tiers changes in the right proportion required to serve the throttled workload volume passed on by the controlled tier. This automatically maintains the optimal power sharing proportion.

An experimental illustration of the pipeline assumption is shown in Figure 3, using a two tier application described in Section 3.5. As the power usage of the controlled tier is reduced, the power usage of the uncontrolled tier changes as well. The figure also shows that the ratio of power consumptions is not constant at different power levels and further depends on the lower layer power control knob used (the figure shows two different DVFS levels), implying that learning a model for this relationship would be non-trivial.

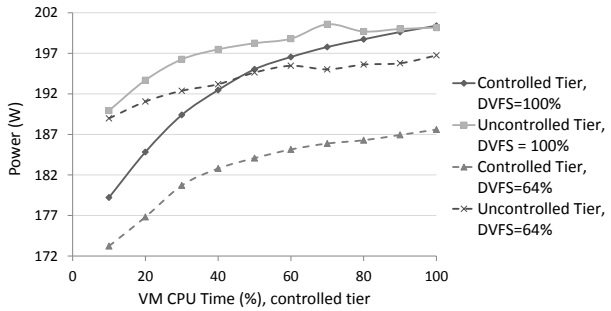


Figure 3: Server power variation with throttling of the controlled tier VM CPU time.

While the pipeline assumption holds in many practical cases and is used in our description, theoretically, the only assumption required is controllability, which is a less stringent requirement. The analysis of controllability conditions is beyond the scope of this paper.

The specific feedback controller used here is based on proportional-integral-derivative (PID) control (Figure 4), governed by:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t) \qquad (3)$$

where $u(t)$ represents the change in power allocated to the controlled tier, $e(t)$ represents the difference between the desired application power budget and the currently measured application power consumption, and the parameters K_p, K_i and K_d are PID controller parameters for the proportional, integral, and derivative terms respectively. The tuning of the controller parameters follows known control theoretic methods and is discussed in Section 3.5.

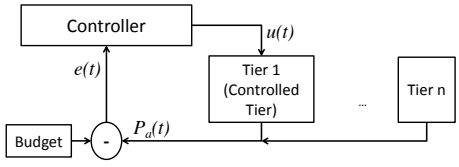


Figure 4: Application level feedback controller.

Conceptually, any of the tiers in the pipeline can be used as the controlled tier. From a practical standpoint, selecting the tier with the largest power variation is desirable as that will provide finer control over the power consumption, leading to lower error in tracking the power budget. The controlled tier can be selected by recording the power variation changes with workload from the power readings provided by each tier.

3.4 Tier Level Controller

The tier level controller, at the controlled tier, controls each of its VMs to track the tier power budget. It communicates with the servers hosting the tier's VMs, to actuate the power control knobs.

The control algorithm at this tier may have multiple power control knobs at its disposal. In our prototype, we use two knobs: VM CPU time allocation and DVFS.

VM CPU time allocation controls the maximum CPU time allocated to a VM, and the processor can enter low power sleep states (also known as C-states) for the unallocated time, reducing the CPU utilization and power consumption [11]. This knob can control the power consumption of an individual VM without affecting other VMs sharing the same processor.

DVFS controls the processor frequency (P-state) to scale CPU power. This knob affects all CPU cores supplied from a single power rail, and thus impacts all VMs sharing those cores.

While we use only CPU-based power knobs, these indirectly influence other components such as the storage subsystem by limiting the workload volume processed and in our experiments we found that power consumption of the storage intensive database tier does vary with CPU power scaling. However, in the future, if the storage subsystem provides direct power control knobs those can be directly used in the VPS framework.

Performance Optimization: Use of multiple power control knobs opens up the opportunity to affect performance. Figure 5 shows the performance of one of the application VMs (StockTrader application, Section 4.1), with different settings of the two knobs. Different types of marks correspond to different DVFS levels while multiple marks of the same type correspond to different VM CPU time allocations at one DVFS setting. The key observation is that a given power level may be achieved at multiple combinations of the two control knobs, yielding different performance levels[3].

[3]The absolute power variation here is small compared to typical server power because the graph only shows the power variation of one VM, and the range of power is restricted to the changes in power of one core. Power is measured in hardware with only one VM allocated. Only change in power is shown.

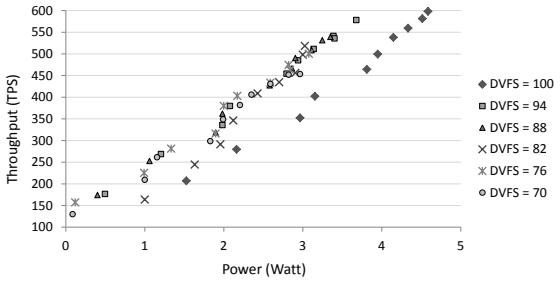


Figure 5: Performance vs power at different DVFS and VM CPU time allocation combinations.

One of the challenges here is to select the power settings for optimal performance. Another challenge is to coordinate the use of the hardware level DVFS knob with power budgeting at application and VM boundaries. A practical constraint for VPS design is that an application performance model and a real time measurement of performance may not be available. This is often the case when data center power and the hosted applications are managed by different entities, and is especially true for cloud platforms.

We study three design options for the tier level controller, making different trade-offs in terms of performance achieved and implementation constraints. All of these designs assume that VMs within a tier are largely homogeneous and load balanced, though they may have small instantaneous variation in their activity.

3.4.1 Open Loop Control

The open loop design assumes that a power model relating the power consumption to the power control knob setting is available for the server hardware. Such models could be learned in-situ using known methods [8]. For instance, if the power knob is VM CPU time, this model may be represented as:

$$P_{VM} = c_{freq} * u_{cpu} \quad (4)$$

where P_{VM} denotes VM power, u_{cpu} is the CPU utilization of the VM, and c_{freq} is a processor frequency dependent power model parameter. Any single control knob can be handled similarly. Multiple simultaneous knobs are considered in Section 3.4.3.

No visibility into application performance is assumed. Only the VM CPU time allocation knob, that acts at VM granularity, is used. VM power allocation is obtained by uniformly dividing the tier power among the tier VMs. For each VM, the assigned power is converted to VM CPU time allocation using (4). The controller is easy to implement and acts instantaneously, but it does not compensate for errors in the model equation (4).

3.4.2 PID Control

Accuracy of the open loop controller can be improved using feedback. Since a feedback controller uses real time power measurements to tune the power setting, it can in fact work even without a power model. VPS uses a PID controller (Figure 6), with one variation that the control output is sent to multiple homogeneous VMs. The control output, $u(t)$, is the VM CPU time allocated to each VM and is assumed to be the same across all VMs within the tier. Small instantaneous differences in VM activity are acceptable since the controller uses only the sum of the powers of all VMs as feedback (small VM variations are averaged out), but the overall VM CPU time to power relationship must be similar for all VMs, implying that a common hardware configuration is used and the software running is the same (since different software functionality can lead to different power consumption even at the same CPU utilization [8]).

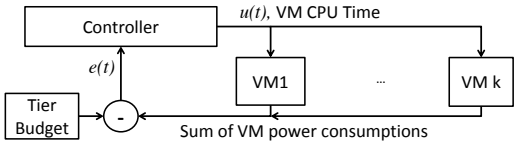


Figure 6: Tier level controller PID feedback loop.

The PID based design provides the advantage of accurate power control. However, since it relies on feedback measurements to reach the desired setting (i.e., tier power budget), it is slower than an open loop control leading to longer control intervals at higher levels of the hierarchy. Also, since it manipulates a single variable, it does not incorporate any notion of optimality for application performance.

3.4.3 Model Predictive Control

The third design option used is based on model predictive control (MPC). MPC allows computing an optimal setting among multiple power control knobs. The optimum is defined in terms of application performance, and hence this design option requires a mechanism to measure application performance.

The cost function optimized by MPC typically consists of two terms: an error term that quantifies the difference from the desired state, and a performance term. The MPC objective function includes not just the current time step but the system state at future time steps, requiring a system model that relates the control knobs to the system state, in this case the target power level and performance. At each time step, the controller solves for the optimal outputs for the next N time steps, applies the solution for only the next time step, and repeats the process to ensure smooth convergence to the desired state.

Cost Function: Suppose $dvfs(i)$ denotes the DVFS, and $v(i)$ denotes the VM CPU time limit, at time step i for a VM. Suppose f_{power} denotes the power model, e.g., equation (4), and f_{perf} the performance model. The cost function, J, is:

$$J = \sum_{i=1}^{N} ||f_{power}(dvfs(i), v(i)) - P_{VM}|| + w \sum_{i=1}^{N} ||f_{perf}(dvfs(i), v(i)) - \alpha_{max}|| \quad (5)$$

where P_{VM} is the VM power to be tracked, α_{max} is the maximum possible performance, and w is a weight that determines the relative importance of the two terms. The first term optimizes the error between the target and predicted power levels, and the second term optimizes performance along the predicted N step control trajectory.

The cost function is minimized to find $dvfs(i)$ and $v(i)$ for best performance. The optimization is solved individually for each VM to keep the size of the optimization search space independent of the number of VMs, ensuring scalability to applications with large number of VMs within a tier.

Hardware Coordination: A block diagram of the control system is shown in Figure 7. The DVFS knob acts at the hardware level and may not respect VM boundaries. Thus, the settings computed above are not applied directly but through a *coordination service*, hosted at each physical server. The service receives DVFS requests from the MPC output for each VM on the server (potentially belonging to different applications) and sets the server DVFS to the highest frequency among the DVFS levels requested. This ensures that no VM is unduly throttled down. The applied DVFS level is reported back. The MPC controllers that receive back a different DVFS setting than the one requested, solve their optimization problem again, with the reported DVFS setting added as a constraint, tuning the VM CPU time knob for the current DVFS setting. The process is repeated at each control iteration yielding the combination of DVFS and VM CPU time allocations that maximizes performance within hardware sharing constraints.

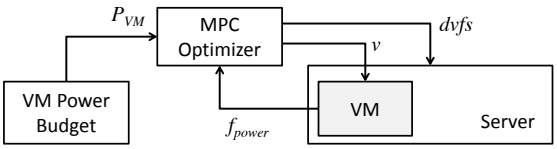


Figure 7: Tier level controller MPC block diagram.

While the MPC based design can yield higher performance than the previous two options, it requires the application performance to be exposed to VPS. Certain cloud platforms such as Microsoft Azure do provide APIs for applications to expose custom performance counters and can be used when available. Power and performance models are also needed. A third consideration is that while the PID controller will always provide a best effort solution within the range of the power settings available, the optimization step in MPC can fail if the optimization is infeasible, and a backup control method may have to be employed. Table 1 summarizes the pros and cons of the above design options.

	Pros	Cons
Open Loop	Fast	Needs power models Higher error
PID	Low error	No performance optimization Slower
MPC	Optimizes performance	Needs system models Needs performance measurement

Table 1: Summary of controller design options.

3.5 Implementation

VPS controllers are implemented as network services on the same physical servers as running the workload. The tier level controller also runs a service in the privileged VM (root VM in Windows Hyper-V) on each physical server to actuate the VM CPU time allocations and DVFS. The network services implementing the controllers also log power and performance data for the experiments. The various parameters and system models needed in the implementation as acquired as follows.

Controller Parameters: The feedback controllers used in the implementation are tuned using known methods from control system design literature. For the PID controllers employed at various layers, the parameters K_p, K_i, and K_d are tuned using the Ziegler-Nichols method [25], on test runs with one of our applications. This method is known to yield robust parameters, keeping the controller stable as workloads change. However, this method does not necessarily yield the fastest convergence or minimum overshoot. Other tuning heuristics available for control system design may be employed as desired. The MPC controller is tuned to operate with a prediction horizon of $N = 1$. Longer time horizons are helpful for ensuring smoother convergence. In VPS, the MPC control is applied only at the individual VM level, where the models are relatively accurate, and hence a short time horizon suffices. The optimization effectively uses the error term as a constraint and maximizes the performance, implying a weight factor w that emphasizes

accurate power tracking over application performance. The detailed optimization and tuning of controller parameters is beyond the scope of this work.

Power and Performance Models: For MPC, the performance model $f_{perf}(dvfs(i), v(i))$ is learned using a test run where each $dvfs(i)$ and $v(i)$ setting is exercised. For each application, this is simply represented as a table with the performance metric listed at each DVFS and VM CPU time setting of the controlled tier. Only a few discrete DVFS levels are available in hardware, and for VM CPU time, nine discrete levels ranging from 10% to 100% are measured. The power model f_{power} is learned using the methods from [8] and is represented using an equation of the form (4). These models need to be learned for each application only if the MPC design option is used. The models depend on hardware used. Large data centers typically have a large numbers of servers for each configuration, and servers are updated in bulk with a single configuration. This means that the models have to be learned on a small number of servers and updated only incrementally.

Additionally, the server infrastructure provides for measuring the total power (from the circuits supplying the servers, cooling, and network equipment). The root VM in each server implements the VM power measurement technique from [8]. The maximum power allowed for each application, denoted P^0_{Ai}, is assumed known and may be determined using the technique described in Section 3.2.1. If multiple applications have different priorities, these are assumed known. In practice, customer facing interactive applications may be assigned one priority level, and batch processing tasks such as MapReduce jobs, data mining, test and development, and internal enterprise applications, could be assigned a second, lower, priority level.

Coordination Across Levels: The controllers at multiple levels are coordinated by setting the control interval of the higher layer controllers to be larger than the convergence time of the lower layer ones. This ensures that the lower layer controller has converged before the higher layer controller receives feedback and actuates, thus avoiding instability. In our prototype, we found the lowest layer controller, when using PID, has a convergence time of 6 seconds and hence, the application level controller uses 6 seconds as its control interval. The application level controller also uses 6 control steps to converge, leading to a control interval of 36 seconds at the data center layer controller. The control algorithm at each layer updates its output at the assigned control interval.

4 Evaluation

4.1 Workloads and Experiment Setup

We use two types of applications for our experiments – an interactive multi-tier application that represents online services subjected to variable user workload, and a set of computationally-intensive batch processing tasks:

StockTrader: StockTrader [17] is an open source multi-tier clustered web application benchmark that mimics a stock trading website, provided for Windows platforms. It is functionally and behaviorally equivalent to IBM WebSphere Trade 6.1 benchmark that runs on other platforms. The application has two significant tiers: a middle tier that implements business logic and a database tier that provides the storage backend. The front-end is a lightweight presentation layer. The incoming requests can be load balanced among multiple VMs hosting the application.

We modified the workload generator provided with the Stocktrader source code to generate workload volume based on a trace file. The application reports its performance in a graphical user interface that we modified to expose the performance as a performance counter sent to the relevant network services implementing the control algorithms in our experiment.

SPEC: We use multiple applications from the SPEC CPU 2006 benchmark suite [15] to represent background jobs that would typically run with lower priority in a data center.

To simulate realistic workloads that vary with time, we use real world data center traces from Windows Live Messenger, an online service with millions of users worldwide. Sample traces from two of its servers are shown in Figure 8. Each instance of the StockTrader application was loaded using a separate data center trace. While the StockTrader application is different from Live Messenger, generating load proportional to production traces helps simulate realistic variations in workload volume.

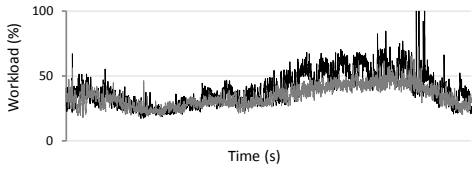


Figure 8: Windows Live Messenger workload trace.

Our testbed consists of seventeen servers, eleven of which host the applications and are subjected to VPS, while the others generate user workload. These are quad core HP ProLiant servers, virtualized using Windows Hyper-V.

Application Deployment: The testbed hosts 4 clustered applications: 3 instances of StockTrader labeled A, B, and C, and a SPEC CPU task set. StockTrader A and StockTrader B are each composed of 13 VMs. StockTrader C is composed of 6 VMs, and the SPEC CPU task set is given 10 VMs. Each VM is assigned one core on one of the quad core servers. StockTrader VMs are allocated to multiple tiers such that each tier reaches high resource utilization at peak load.

Stocktrader B and C are treated as high priority applications while StockTrader A and SPEC are given low priority. The VMs are mixed up across servers such that some servers host both high and low priority VMs while others host VMs only from a single priority level. Each server hosts VMs from more than one application.

Measurements: Power measurements for the hosted VMs are obtained using [8]. This technique obtains a mapping between resource usage, which can be monitored for each VM by the hypervisor, to actual VM power use, since VM power cannot be measured in hardware. Power measurements for the entire testbed are obtained in hardware, using a set of WattsUp PRO [24] meters, connected to each of the servers. This hardware measurement includes the base power consumption of the servers (power consumed when powered on but idle) that is not attributed to any specific VM, and is treated as $P_M(t)$ for equations (1) and (2). Cooling equipment is not part of this testbed. When using MPC at the tier level, the SPEC application's tier level controller is still PID, because the SPEC CPU applications does not expose performance metrics in real time.

Comparison: In addition to the VPS controllers with multiple options from Table 1, we also implemented a power budgeting system that simply follows the hierarchy of the physical layout of the testbed, for comparison. This controller uses only DVFS as its power control knob and operates at the server level, similar to prior works [22]. Servers that are exceeding their allocated budget, i.e., the ones with highest resource usage, are throttled first.

Illustrative Run: We conduct multiple runs with different workload traces and take an average of the measured metrics (5 runs in each experiment). As an illustration, Figure 9 shows the power consumption with and without VPS controllers, for part of a run. Tracking is enforced during time intervals where uncontrolled consumption (dashed line) is above the tracked power level (solid black line). Only two of the controllers are shown for clarity. The controllers do exceed the tracked power level at times, leading to tracking errors. Also, even when the uncontrolled curve exceeds the tracked power level, implying that the workload is high, the controllers sometimes leave power unused below the tracked level, taking an unnecessary performance hit.

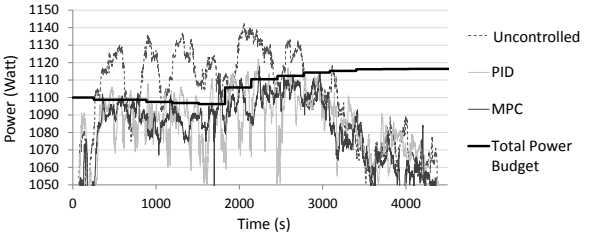


Figure 9: A workload run against various control systems with dynamic power budget.

The figure also illustrates the use of a time varying power capacity availability, as may be useful in scenarios where the utility power is supplemented with environmentally harvested power [16, 2].

4.2 Results

The performance metrics of interest are: total (data center level) and application level *power budgeting errors*, application performance *differentiation* (ability to operate interactive applications on shared infrastructure with low priority tasks), and *performance* achieved within the power budget.

4.2.1 Power Budgeting Errors

Error is defined as the excess power consumed above the assigned power budget, normalized by the power budget:

$$TrackingError = MAX\left\{\frac{P_M(t) - P_T(t)}{P_T(t)}, 0\right\}$$

where $P_M(t)$ represents the measured data center power consumption. Consumption below the target level may result from workload being low or the controller being overly conservative. Being overly conservative is not an error from a budgeting perspective, but penalizes the controller in terms of achieved application performance.

Figure 10 shows the average and standard deviation of the mean error across all experiment runs, for each design choice. The PID-based system has higher error because the PID controller has higher overshoots during its convergence time, compared to MPC and Open Loop systems, and is as expected. Higher oscillations for PID compared to MPC were also seen in [21]. The physical hierarchy based controller has higher error primarily because the control knob it uses, DVFS, is not as fine grained as VM CPU time allocation. Processors have only a few discrete DVFS levels as opposed to CPU time allocation that can varied in fine grained steps. Overall however, each of the design choices yields fairly low error and the choice will thus depend on the other implementation constraints or performance considerations.

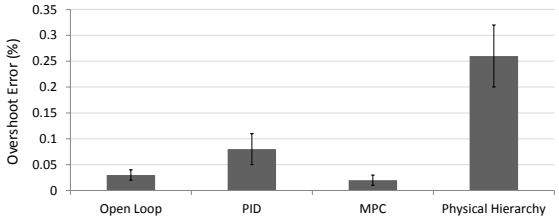


Figure 10: Total power errors of each control system.

It is worth noting that the overall error is low even when using open loop control, because some of its error is compensated by higher layer controllers. The error in open loop control is more apparent at the lower layers. Figure 11 shows the mean error for each hosted application (ST-x refers to StockTrader-x). Here, the PID and MPC based systems have similar application power errors, and both fare better than the open loop VPS system. ST-A and SPEC being the lower priority applica-

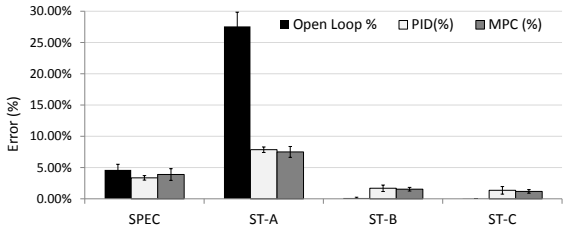


Figure 11: Application power enforcement errors.

tion are subject to greater power reduction. However, the open loop controller has a poorer power model for StockTrader applications than SPEC (the power model accuracy can vary across applications since different applications may use resources differently [8]). As a result, it underestimates the power consumption of ST-A, and does not throttle it sufficiently, leading to high error. Due to this error, the higher layer controller reduces the total power available to all applications, resulting in the higher priority applications, StockTrader B and C, seeing lower budgets under open loop design than their actual limits in other designs. These are throttled more than necessary and stay well below the target level, resulting in tracking error being virtually eliminated for B and C. The physical hierarchy based controller does not apply to individual applications and is omitted in this figure.

4.2.2 Power Differentiation

VPS is designed to respect application priorities and QoS constraints in a shared infrastructure. Figure 12 shows the differentiation between different applications enabled by VPS. Power reduction compared to uncontrolled operation is shown, normalized by the uncontrolled consumption.

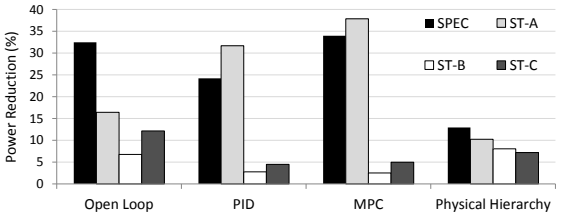


Figure 12: Average application power reduction under each control system. SPEC and ST-A are the lower priority applications and ideally only these two should have their power reduced.

The physical hierarchy based controller, which operates at the hardware level without consideration of application VM boundaries, is unable to differentiate between applications: higher and lower priority applications have their power reduced by similar amounts. In contrast, the PID and MPC based VPS systems show marked application power differentiation.

The open loop system does differentiate, and SPEC is throttled by similar amount as with MPC and PID. However, the power model used does not work as well for the StockTrader applications and we see that StockTrader A is throttled much less, causing the higher priority applications to be throttled more, as explained with Figure 11.

4.2.3 Application Performance

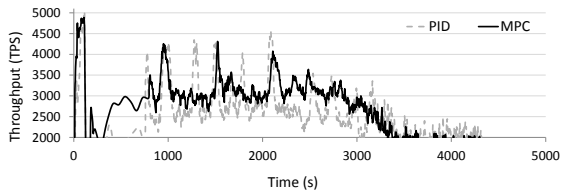
We saw above that both the PID and MPC based VPS systems can perform appropriate application differentiation and achieve low errors. The distinguishing feature of MPC however, is its ability to improve application performance by intelligently selecting the appropriate combination of power settings that yields higher performance for a given power level.

An illustration of this effect is shown in Figure 13, which shows the throughput and response time achieved by StockTrader A, under both MPC and PID based approaches, for the same power budget. MPC yields higher throughput and lower response times, showing a noticeable performance advantage.

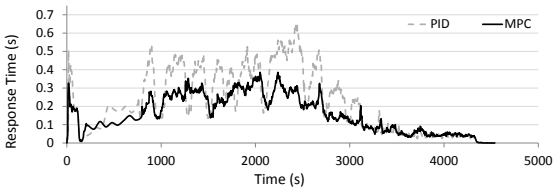
Quantitatively, the performance difference is measured as follows. The degradation, δ, in performance is defined as the fractional reduction in performance compared to when run with unlimited power:

$$\delta = \left| \frac{Perf_{unlimited} - Perf_{VPS}}{Perf_{unlimited}} \right|$$

for both response time and throughput. For each experiment run, we calculate the mean degradation for each application. The degradations in throughput and response time are compared in Figure 14. In each case, the MPC based system shows lower performance degradation, implying higher performance. StockTrader B and C being



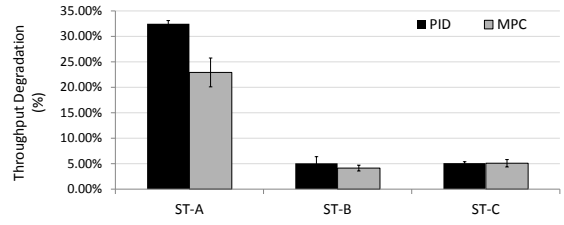
(a) Application Throughput.



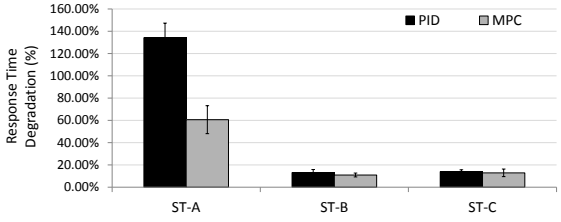
(b) Application Response Time.

Figure 13: Application performance of a low priority (throttled) application, for MPC and PID, with the same power budget.

higher priority applications, are not affected much in either PID or MPC, but StockTrader A shows a marked performance advantage for using MPC.



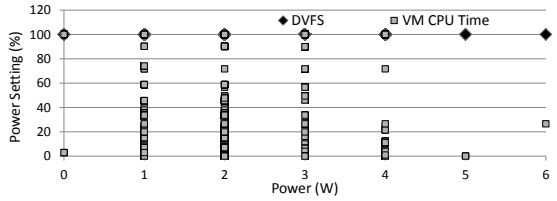
(a) Throughput Degradation



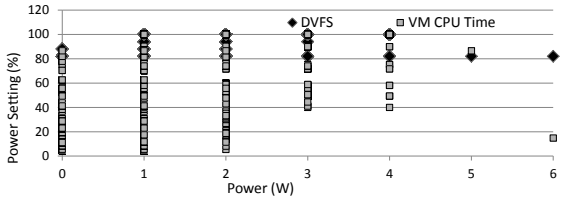
(b) Response Time Degradation

Figure 14: Performance degradation: MPC vs PID. The high priority applications (ST-B and ST-C) are not throttled much but ST-A suffers lower degradation when throttled using MPC than with PID for the same power budget.

We also noted in Section 3.4.3 that MPC is restricted in its use of the DVFS control knob because the hardware is shared among multiple VMs. While the VM CPU time limit can be applied to any VM, the DVFS knob can only be used when allowed by the coordination service. To study the impact of coordination, we track the DVFS and VM CPU time limit settings used at two of StockTrader A's VMs. One of these VMs, labeled VM1, is co-located with VMs from higher priority applications while the other, VM2, is placed on a server where all other VMs belong to lower priority applications. Figures 15(a) and 15(b) show the power control knob settings used by these two VMs during the MPC experiment, at different times during an entire run. We see that VM1 is unable to use the DVFS knob (DVFS is always 100%) because the other VMs on that server require the highest DVFS setting. VM2 on the other hand does use multiple DVFS levels, and its spread of VM CPU time limits is thus different from VM1.



(a) VM1 power settings



(b) VM2 power settings

Figure 15: Power control knob settings used for VM1 (co-located with high priority application VMs) and VM2 (co-located with low priority application VMs).

The performance advantage shown by MPC in Figures 14(a) and 14(b) is obtained using the DVFS knob only in cases where the VM placement allowed its use. The performance advantage may be higher in scenarios where all low priority VMs do not share servers with higher priority VMs.

5 Discussion and Future Work

VPS enables performance aware power budgeting in multi-application virtualized scenarios. The system can be extended to incorporate additional features and application scenarios, as follows.

Server Shutdown: In this paper, we only used two power control knobs: DVFS and VM CPU time allocation. Both these knobs can be applied in real time with low latency. However, these knobs only influence the portion of server power consumption that varies with processor power settings. A significant portion of server power, as high as 50-60%, is spent to simply power up a server, and is referred to as idle power. Therefore, an effective means to reduce power is to shut down some

servers, instead of throttling down servers that are powered up. Consider as a toy example, a set of 10 servers, each of which consumes 50W at idle and 100W at peak load (i.e., server power increases from 50W to 100W as CPU utilization increases from 0 to 100%). The total power consumption is 1000W at peak load. Suppose a reduction of 250W is desired. One option is to reduce the CPU utilization of each server from 100% to 50%, reducing the power level of each server to 75W. The number of CPU cycles available is reduced by 50% in this case. Another option is to shut down three of the servers, reducing the power by 300W but reducing the CPU cycles by only 30% (instead of 50%). The second approach achieves part of its power reduction by eliminating idle power of three servers and can offer higher performance.

Clearly, exploiting server shutdown as a power control knob has a performance advantage. However, server shutdown has high latency. Also, since it is a hardware level knob, coordination among all VMs located on a server to be shut down is required. Commercial products that can automatically migrate or shutdown VMs and servers as resource utilization changes are already available [20]. Incorporating such techniques into application power budgeting presents interesting research challenges and will likely yield significant performance benefits when power throttling is required for longer time durations.

Additional Applications: We explained the choice of control intervals in Section 3.5. The latencies achieved are acceptable for over-subscription with a static power capacity, since the only dynamics come from user workloads and these vary only gradually over the course of a day [1]. Such latencies are also acceptable for additional power budgeting scenarios. VPS may be used to control power usage in a multi-application server cluster powered wholly or in part from environmentally-harvested energy, such as solar power, since it varies relatively slowly. VPS may also be employed when the data center wishes to change its power budgets with demand response based or time of day based power prices. The power prices are adjusted for periods of at least an hour or 30 minutes in most electricity markets, allowing ample time for controller convergence.

6 Related Work

Several prior works have considered the problem of power and performance control of data center servers. Power budgeting for a single server has been considered in [9]. Multi-server power budgeting, sometimes referred to as power shifting, has also been discussed [23, 21, 3, 22]. The power controllers proposed in [21, 23] distribute power proportional to CPU utilization in a cluster of servers. In [3], workload differences among multiple nodes are used to allocate different power limits. In [22], a hierarchical approach is used where the controller hierarchy follows the physical server and rack layout. In all these works, only DVFS is used as the power control knob and application differentiation is not considered. The use of multiple power control knobs is also not considered. The controller centrally measures and actuates each server, limiting scalability. Optimal allocation of available power to maximize performance was also considered in [4], where a choice was made between the number of active servers and their processor frequencies. A single application running on homogeneous servers was considered and power allocations were made centrally. We extend the above works to allow multiple applications sharing a common server infrastructure. We also design a method to select an optimal combination of power settings when multiple options exist for achieving the same power level, rather than always using DVFS. We further adapt power allocations dynamically across applications and application tiers to improve performance.

The performance of multi-tier applications has also been considered in [7, 10]. The method in [7] tunes the power settings at each tier to meet an overall performance objective by determining coordinated frequency levels for each tier. In [10], one controller is used to set the processor frequency of one of the tiers to meet performance requirements and another controller tunes the frequency of the other tier to minimize overall energy. These methods require detailed performance models across multiple tiers. We optimize performance across multiple tiers using low overhead mechanisms that do not require learning multi-tier performance models. Our methods work with dynamic workloads and can also use multiple power control knobs.

Partitioning of power due to limitations of power distribution may also lead to inefficient operation because unused power capacity in one part of the data center cannot be delivered to other parts. Solutions to this problem have been discussed before [13]. We assume that such solutions have been deployed, and the distribution infrastructure is not a limiting factor.

In addition to the above works, several others have addressed various related aspects of power control. Coordination of multiple controllers for joint objectives of power capacity, energy consumption, and thermal management was presented in [14]. Our solution addresses multi-application scenarios with dynamic workloads and application performance optimization. Design time analysis of coordinated controllers for detecting unwanted positive feedbacks and instability was presented in [6]. We use multiple coordinated controllers in a hierarchy such that they do not lead to positive feedbacks and ensure stability through known methods. Design time

methods for efficient power provisioning, based on statistical profiling, have also been studied [5] but are complementary to our work. VPS power tracking methods are designed to be employed at run time, after the design time provisioning limits have been determined. Modeling and control of application performance and resource usage in a virtualized infrastructure has also been considered before [12]. Our focus is specifically on power tracking, with appropriate mechanisms for performance differentiation and optimization.

We also consider several practical aspects not previously considered. For instance, the platform providing and controlling the power limits has very limited visibility into the application performance metrics. This is especially true for cloud environments where the applications may not be owned by the same entity that manages the data center and its power usage. Further, the workload for one application may change, causing the power availability for other applications to change and hence we dynamically adapt to such changes. We also do not assume that detailed models for power distribution across multiple application tiers can always be learned.

7 Conclusion

We presented a power budgeting system, VPS, for virtualized data centers hosting multiple applications. VPS can significantly improve the power capacity utilization by providing effective power budgeting in multiple scenarios including over-subscription, energy harvesting data centers, and variable power pricing. VPS allocates available power efficiently among multiple applications sharing the same servers and adapts to dynamic workload variations. The pipelined organization of large scale applications into tiers is used to automatically distribute power among the application tiers in appropriate proportions. Multiple power control knobs are exploited for optimizing performance. The algorithms used are based on control theoretic techniques to help ensure stable and robust operation. VPS offers multiple implementation options to adapt to practical design constraints such as lack of detailed system models and limited visibility into application performance.

8 Acknowledgments

The authors are grateful to Prof Xenofon Koutsoukos (Vanderbilt University) for his insightful comments.

References

[1] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI* (2008), pp. 337–350.

[2] CLIDARAS, J., STIVER, D. W., AND HAMBURGEN, W. Water-based data center. US Patent Application, February 2007.

[3] E. FEMAL, M., AND W. FREEH, V. Boosting data center performance through non-uniform power allocation. In *ICAC* (2005).

[4] GANDHI, A., HARCHOL-BALTER, M., DAS, R., AND LEFURGY, C. Optimal power allocation in server farms. In *SIGMETRICS* (2009).

[5] GOVINDAN, S., CHOI, J., URGAONKAR, B., SIVASUBRAMANIAM, A., AND BALDINI, A. Statistical profiling-based techniques for effective power provisioning in data centers. In *EuroSys* (2009).

[6] HEO, J., HENRIKSSON, D., LIU, X., AND ABDELZAHER, T. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *RTSS* (2007).

[7] HORVATH, T., ABDELZAHER, T., SKADRON, K., AND LIU, X. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *IEEE Trans. Comput. 56*, 4 (2007), 444–458.

[8] KANSAL, A., ZHAO, F., LIU, J., KOTHARI, N., AND BHATTACHARYA, A. A. Virtual machine power metering and provisioning. In *SoCC* (2010).

[9] LEFURGY, C., WANG, X., AND WARE, M. Server-level power control. In *ICAC* (2007).

[10] LEITE, J. C., KUSIC, D. M., AND MOSSÉ, D. Stochastic approximation control of power and tardiness in a three-tier web-hosting cluster. In *ICAC* (2010).

[11] NATHUJI, R., ENGLAND, P., SHARMA, P., AND SINGH, A. Feedback driven qos-aware power budgeting for virtualized servers. In *FeBID* (2009).

[12] PADALA, P., HOU, K.-Y., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., AND MERCHANT, A. Automated control of multiple virtualized resources. In *EuroSys* (2009).

[13] PELLEY, S., MEISNER, D., ZANDEVAKILI, P., WENISCH, T. F., AND UNDERWOOD, J. Power routing: dynamic power provisioning in the data center. In *ASPLOS* (2010).

[14] RAGHAVENDRA, R., RANGANATHAN, P., TALWAR, V., WANG, Z., AND ZHU, X. No "power" struggles: coordinated multi-level power management for the data center. In *ASPLOS* (2008).

[15] SPEC CPU2006. `http://www.spec.org/cpu2006`.

[16] STEWART, C., AND SHEN, K. Some joules are more precious than others: Managing renewable energy in the datacenter. In *HotPower* (2009).

[17] .NET StockTrader Sample Application. http://msdn.microsoft.com/en-us/netframework/bb499684.aspx.

[18] STOESS, J., LANG, C., AND BELLOSA, F. Energy management for hypervisor-based virtual machines. In *USENIX Annual Technical Conference* (2007).

[19] URGAONKAR, B., SHENOY, P., AND ROSCOE, T. Resource overbooking and application profiling in a shared internet hosting platform. *ACM Trans. Internet Technol. 9*, 1 (2009), 1–45.

[20] VMWARE. VMware distributed power management concepts and use. http://www.vmware.com/files/pdf/DPM.pdf.

[21] WANG, X., AND CHEN, M. Cluster-level feedback power control for performance optimization. In *HPCA* (2008).

[22] WANG, X., CHEN, M., LEFURGY, C., AND KELLER, T. W. Ship: Scalable hierarchical power control for large-scale data centers. In *PACT* (2009).

[23] WANG, X., AND WANG, Y. Coordinating power control and performance management for virtualized server clusters. *IEEE Transactions on Parallel and Distributed Systems 99* (2010).

[24] WattsUp PRO ES. https://www.wattsupmeters.com/secure/index.php.

[25] ZIEGLER, J., AND NICHOLS, N. B. Optimum settings for automatic controllers. *Transactions of the ASME 64* (1942), 759–768.

vIOMMU: Efficient IOMMU Emulation

Nadav Amit
Technion & IBM Research

Muli Ben-Yehuda
Technion & IBM Research

Dan Tsafrir
Technion

Assaf Schuster
Technion

Abstract

Direct device assignment, where a guest virtual machine directly interacts with an I/O device without host intervention, is appealing, because it allows an unmodified (non-hypervisor-aware) guest to achieve near-native performance. But device assignment for unmodified guests suffers from two serious deficiencies: (1) it requires pinning all of the guest's pages, thereby disallowing memory overcommitment, and (2) it exposes the guest's memory to buggy device drivers.

We solve these problems by designing, implementing, and exposing an emulated IOMMU (vIOMMU) to the unmodified guest. We employ two novel optimizations to make vIOMMU perform well: (1) waiting a few milliseconds before tearing down an IOMMU mapping in the hope it will be immediately reused ("optimistic teardown"), and (2) running the vIOMMU on a sidecore, and thereby enabling for the first time the use of a sidecore by unmodified guests. Both optimizations are highly effective in isolation. The former allows bare-metal to achieve 100% of a 10Gbps line rate. The combination of the two allows an unmodified guest to do the same.

1 Introduction

I/O activity is a dominant factor in the performance of virtualized environments [29, 37], motivating *direct device assignment* whereby a guest virtual machine (VM) sees a real device and interacts with it directly. As direct access does away with the software intermediary that other I/O virtualization approaches require, it can provide much better performance than the alternative I/O virtualization approaches. This increased performance comes at a cost of complicating virtualization use-cases where the hypervisor interposes on guest I/O, such as live migration [20, 50]. Nonetheless, the importance of increased I/O performance cannot be overstated, as it makes virtualization applicable to common I/O-intensive workloads that would otherwise experience unacceptable performance degradation [26, 28, 33, 45, 48].

1.1 Motivation

Despite its advantages, direct device assignment suffers from at least three serious deficiencies that limit its applicability. First, it requires the entire memory of the unmodified guest to be pinned to the host physical memory. This is so because I/O devices typically access the memory by triggering DMA (direct memory access) transactions, and those can potentially target any location of the physical memory; importantly, unlike regular memory accesses, computer systems are technically unable to gracefully tolerate DMA page misses, reacting to them by either ignoring the problem, by restarting the offending domain, or by panicking. The hypervisor cannot tell which pages are designated by the unmodified guest for DMA transactions, and so, to avoid such unwarranted behavior, it must pin all the guest's pages to physical memory. This necessity negates a primary reason for using virtualization—server consolidation—because it hinders the ability of the hypervisor to perform memory overcommitment, whereas memory is *the* main limiting factor for server consolidation [16, 41, 47].

The second deficiency of direct device assignment is that the unmodified guest is unable to utilize the IOMMU (I/O memory management unit) so as to protect itself against bugs in the corresponding drivers. It is well-known that device drivers constitute the dominant source of OS (operating system) bugs [5, 17, 25, 38, 43]. Notably, the devices' ability to perform DMA to arbitrary physical memory locations is a main reason why such bugs are detrimental. IOMMUs were introduced by all major chip manufacturers to solve exactly this problem. They allow the OS to restrict DMA transactions to specific memory locations by having devices work with IOVAs (I/O virtual addresses) instead of physical addresses, such that every IOVA is validated by the IOMMU hardware circuitry upon each DMA transaction and is then redirected to a physical address according to the IOMMU mappings. The hypervisor cannot allow guests to program the IOMMU directly (otherwise every guest would be able to access the entire physical memory), and so all the related work that provided ways for guests to enjoy the IOMMU functionality [12, 13, 25, 35, 44, 49] involved paravirtualization. Namely, the guest's OS was modified to explicitly inform the hypervisor regarding the DMA mappings it requires. Clearly, such an approach is inapplicable to unmodified (fully virtualized) guests.

A third deficiency of direct device assignment is that, in general, it prevents the unmodified guest from taking advantage of the IOMMU remapping capabilities, which

are useful in contexts other than just defending against faulty device drivers. One such context is legacy devices that do not support memory addresses wider than 32bit, an issue that can be easily resolved by programming the IOMMU to map the relevant 32bit-addresses to higher memory locations [18]. Another such context is "nested virtualization", which allows one hypervisor to run other hypervisors as guests [11] and, hence, mandates granting a nested hypervisor the ability to program the IOMMU to protect its guests from one another (when those utilize directly-assigned devices).

1.2 Contributions and Preview of Results

IOMMU Emulation The root cause of all of the above limitations is the fact that current hypervisors do not provide unmodified guests with an emulated IOMMU. Our initial contribution is therefore to implement and evaluate such an emulation, for the first time. We do so within KVM on Intel x86, following the proposal made by Intel [1]. We denote the emulation layer "vIOMMU". And we note in passing that we are aware of a similar effort that is currently being done for AMD processors [31].

By emulating the IOMMU, our patched hypervisor intercepts, monitors, and acts upon DMA remapping operations. Knowing which of the unmodified guest's memory pages serve as DMA targets allows it to: (1) pin/unpin the corresponding host physical pages, and only these pages, thereby enabling memory overcommitment; (2) program the physical IOMMU to enable device access to the said physical pages, and only to these pages, thereby enabling the guest to protect its memory image against faulty drivers; and (3) redirect DMA transactions through the physical IOMMU according the unmodified guest's wishes, thereby retrieving the indirection level needed to support legacy 32bit devices, certain user-mode DMA usage models, and nested virtualization. (See Section 2 for details.)

Utilizing the IOMMU without relaxing the protection it offers is costly, even for a bare metal (unvirtualized) OS. Our experiments using Netperf [19] show that bare metal Linux 2.6.35 achieves only 43% of the line-rate of a 10Gbps NIC when the IOMMU is used with strict protection; the corresponding unmodified guest achieves less than one fourth of that with the vIOMMU.

Optimistic Teardown The default mode of Linux, however, relaxes IOMMU protection. It does so by batching the invalidation of stale IOTLB entries and by collectively purging them from the IOTLB every 10ms (IOTLB is the I/O translation look-aside buffer within the IOMMU). The protection is relaxed, because, during this short interval, a faulty device might successfully perform a DMA transaction through a stale entry. Nonetheless, for bare metal, the resulting improvement is dramatic, transforming the aforesaid 43% throughput to 91% and arguably justifying the risk. Alas, the corresponding unmodified guest does not experience such an improvement, as its throughput remains more or less the same when the protection is relaxed.

To improve the performance of the vIOMMU, our second contribution is investigating a set of optimizations that exercise the protection/performance tradeoff in various ways (see Section 3 for details). We find that the "optimistic teardown" optimization is the most effective.

While the default mode of Linux removes stale IOTLB entries en masse at 10ms intervals, it nevertheless tears down individual invalidated IOVA translations with no delay, immediately removing them from the IOMMU page table. The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the OS mapping data structure, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [3, 44], one can be hopeful that the newly proposed optimization would work. Importantly, for each reused translation, optimistic teardown would avoid the overhead of (1) tearing the translation down from the IOMMU page table, (2) invalidating it in the IOTLB, (3) immediately reconstructing it, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Optimistic teardown is remarkably successful, pushing the throughput of bare metal from 91% to 100% (and reducing its CPU consumption from 100% to 60%). The improvement is more pronounced for an unmodified guest with vIOMMU: from 11% throughput to 82%.

Sidecore Emulation To further improve the performance of the unmodified guest, we implement the vIOMMU functionality on an auxiliary sidecore. Traditional "samecore" emulation of hardware devices (where hypervisor invocations occur on the guest's core) has been extensively studied in the literature [6, 10, 21, 37]. Likewise, offloading of computation to a sidecore for speeding up I/O in a paravirtualized system has been explored as well [15, 23, 27]. But in this paper, for the first time, we present "sidecore emulation", which combines the best of both approaches. Specifically, sidecore emulation maintains the exact same hardware interface between the guest and the sidecore as exists in a non-virtualized setting between a bare metal OS and the real hardware device. Consequently, sidecore emulation is able to offload the computation while requiring no guest modifications. (See details in Section 4.)

By running the vIOMMU on a sidecore, we triple the

setting	*strict*	*relaxed (default)*	*optimistic teardown*
samecore	10%	11%	82%
sidecore	30%	49%	100%
bare metal	43%	91%	100%

Table 1: *Summary of preview of results (percent of line-rate throughput on 10GbE).*

throughput of the strict unmodified guest, quintuple its throughput if its protection is relaxed, and achieve 100% of the line-rate if employing optimistic teardown. The results mentioned so far are summarized in Table 1.

Roadmap We describe: our "samecore" vIOMMU design (§2); the set of optimizations we explore and the associated performance/protection tradeoffs (§3); our "sidecore" vIOMMU design (§4); how to reason about risk and protection (§5); evaluation of the performance of our proposals using micro and macro benchmarks (§6); the related work (§7); and our conclusions (§8).

2 Samecore IOMMU Emulation

I/O device emulation for virtualized guests is usually implemented by trapping guest accesses to device registers and emulating the appropriate behavior [2, 10, 37]. Correspondingly, in this section, we present the rudiments of emulating an IOMMU. We choose to emulate Intel's VT-d IOMMU [18], as it is commonly available and as most x86 OSes/hypervisors have drivers for it. Conveniently, Intel's VT-d specification [18] proposes how to emulate an IOMMU. We largely follow their suggestions.

The emulated guest BIOS uses its ACPI (Advanced Configuration and Power Interface) tables to report to the guest that the (virtual) hardware includes Intel's IOMMU. Recognizing that the hardware supports an IOMMU, the guest will ensure that any DMA buffer in use will first be mapped in the IOMMU for DMA [12]. The emulated IOMMU registers reside in memory pages that the hypervisor marks as "not present", causing any guest access to them to trap to the hypervisor. The hypervisor monitors the emulated registers and configures the platform's physical IOMMU accordingly. The hypervisor further monitors changes in related data structures such as the IOMMU page tables in guest memory.

Figure 1 illustrates the flow of a single DMA transaction in an emulated environment: a guest I/O device calls the IOMMU mapping layer when it wishes to map an I/O buffer (1); the layer accordingly allocates an IOVA region and, within the emulated IOMMU, maps the corresponding page table entries (PTEs) to point to the GPA (guest physical address) given by the I/O device driver (2); the layer performs an explicit mapping invalidation of these PTEs (3), thereby triggering a write access to a certain IOMMU register, which traps to the hypervisor; the hypervisor then updates the status of the emulated IOMMU registers (4), reads the IOVA-to-GPA mapping from the updated emulated IOMMU PTEs (5), pins the relevant page to the host physical memory (not shown), and generates physical IOMMU PTEs to perform IOVA-to-HPA (host physical address) mapping (6); when the physical hardware requires it, the hypervisor also performs physical IOTLB invalidation (7); the guest is then resumed, and the I/O device driver initiates the DMA transaction, delivering the IOVA as the destination address to the device (8); the device performs memory access to the IOVA (9), which is appropriately redirected by the physical IOMMU (10-11); the guest OS can then unmap the IOVA, triggering a flow similar to the mapping flow except that the hypervisor unmaps the I/O buffer and unpins its page-frames.

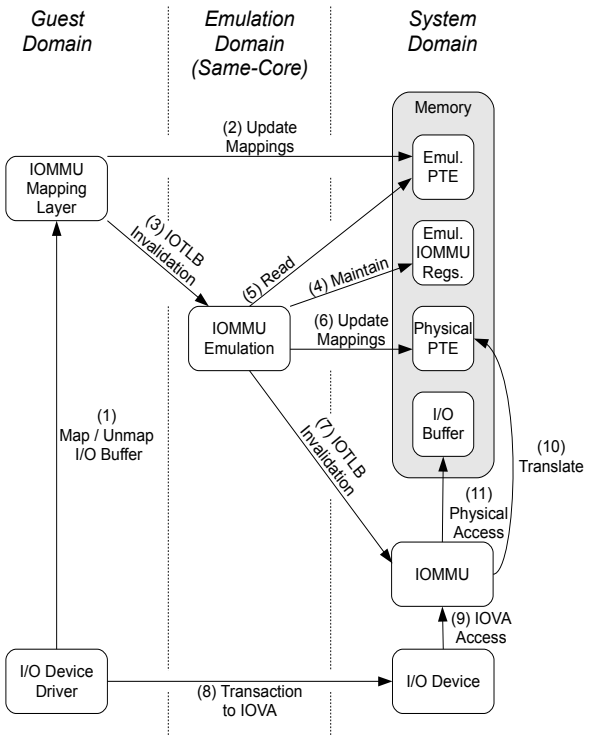


Figure 1: *IOMMU emulation architecture (samecore).*

3 Optimizing IOMMU Mapping Strategies

Operating systems can employ multiple mapping strategies when establishing and tearing down IOMMU mappings. Different mapping strategies tradeoff performance vs. memory consumption vs. protection [13, 44, 49].

Taking Linux as an example, the default mapping strategy of the Intel VT-d IOMMU driver is to *defer* and batch IOTLB invalidations, thereby improving performance at the expense of reduced protection from errant DMAs. Batching IOTLB invalidations helps performance because IOTLB invalidations are expensive. Unlike an MMU TLB, which resides on a CPU core, an IOMMU and its IOTLB usually reside away from the CPU on the PCIe bus.

An alternative mapping strategy is the *strict* mapping strategy. In the strict strategy Linux's IOMMU mapping layer executes IOTLB invalidations as soon as device drivers unmap their I/O buffers and waits for the invalidations to complete before continuing.

In this section we investigate the different tradeoffs possible on bare metal and in a virtualized system employing an emulated IOMMU, where both the guest and the host may employ different mapping strategies. We discuss different IOMMU mapping performance optimizations and their effect on system safety, starting with the least dangerous strategy and ending with the best performing—but also most dangerous—strategy.

3.1 Approximate Shared Mappings

Establishing a new mapping in the IOMMU translation table and later tearing it down are inherently costly operations. Shared mappings can alleviate some of the costs [44]. We can reuse a mapping when another valid mapping which points to the same physical page frame already exists. Using the same mapping for two mapping requests saves the time required for the setup and eventual teardown of a new mapping.

Willmann, Rixner and Cox propose a precise lookup method for an existing mapping. Their approach relies on an inverted data structure translating from physical address to IOVA for all mapped pages [44]. This approach is problematic with modern IOMMUs that can map all of physical memory and employ a separate I/O virtual address space for each protection context (usually for each I/O device). Maintaining a direct-map data structure to enable precise lookups is impractical for such IOMMUs as it would require too much memory. We expect that using a smaller but more complex data structure, such as a red-black tree, will incur prohibitively high overhead [32].

To avoid the overhead associated with complex datastructures, we propose *approximate shared mappings*. Instead of maintaining a precise inverted data structure, we perform reverse lookups using heuristics which may fail to find a translation from physical address to IOVA, even though there exists a mapping of that physical address. Our implementation of approximate shared mappings used a software LRU cache, which requires temporal locality in I/O buffers allocation in order to perform well, in addition to spatial locality of the I/O buffers. Many applications experience such temporal locality [44].

3.2 Asynchronous Invalidations

IOTLB invalidation is a lengthy process that on bare metal takes over 40% of the overall unmapping process. *Asynchronous invalidation* is an invalidation scheme targeted at alleviating the cost of the lengthy IOTLB invalidation process by a minor relaxation of protection. The default IOTLB invalidation scheme is synchronous: the OS writes an invalidation request to the IOMMU's invalidation register or (when the hardware supports it) to an invalidation queue [18] and blocks the execution thread until the IOMMU completes the invalidation. In asynchronous invalidation, the OS does *not* wait for the invalidation to complete before continuing. Doing so on bare metal can save the few hundred cycles it takes the IOMMU to write the invalidation completion message back to memory after the invalidation is done.

Asynchronous invalidation enables multiple in-flight invalidations when the hardware supports an invalidation queue. However, to maintain correctness, asynchronous invalidation must not permit an IOVA range which is being invalidated to be mapped again to a different physical address until the invalidation process is completed. Unfortunately there is no practical way to ensure with Linux that the page allocator will not reuse the physical memory backing those IOVAs while the invalidation is outstanding [49].

On bare metal asynchronous invalidation relaxes protection only slightly, since the IOMMU hardware performs the invalidation process in silicon, taking only hundreds of cycles to complete. In our experiments with asynchronous invalidation, the invalidation queue never held more than two pending invalidations at the same time.

3.3 Deferred Invalidation

Deferring IOTLB invalidations, as currently implemented by Linux, makes it possible to aggregate IOTLB invalidations together and possibly coalesce multiple invalidation requests so that they will be invalidated in a single request, if the hardware supports it. Instead of the OS invalidating each translation entry as it is torn down, the OS collects multiple invalidations in a queue, which it then flushes periodically. The current Linux implementation coalesces up to 250 invalidations for periods of no longer than 10ms.

Holding back the invalidations makes the deferred method less secure than the asynchronous method, where

the "window of vulnerability" for an errant DMA is only hundreds of cycles. But deferred invalidation reduces the number of software/hardware interactions, since a whole batch of invalidations is executed at once. This savings is more pronounced when the hardware is emulated by software, in which case deferred invalidation can save multiple, expensive guest/host interactions.

3.4 Optimistic Teardown

Reusing IOVA translations is key to IOMMU performance [3, 13, 44, 49]. Reusing a translation avoids the overhead of (1) tearing a translation down from the IOMMU page table, (2) invalidating it from the IOTLB, (3) immediately reconstructing it in the page table, and (4) reinserting it back to the IOTLB; all of which are costly operations, as each IOTLB modification involves updating uncacheable memory and teardown/reconstruction involves nontrivial logic and several memory accesses.

Even when approximate shared mapping is used, the opportunities to reuse IOVA translations are limited. The default Linux deferred invalidation scheme removes stale IOTLB entries en masse at 10ms intervals, but nevertheless tears down individual unmapped IOVA translations with no delay, immediately removing them from the IOMMU page tables.

The rationale of optimistic teardown rests on the following observation. If a stale translation exists for a short while in the IOTLB anyway, we might as well keep it alive (for the same period of time) within the IOMMU page table, optimistically expecting that it will get reused (remapped) during that short time interval. As significant temporal reuse of IOVA mappings has been reported [3, 44, 49], one can be hopeful that the newly proposed optimization would work.

We thus developed an optimistic teardown mapping strategy, which keeps mappings around even after an unmapping request for them has been received. Unmapping operations of I/O buffers are deferred and executed at a later, configurable time. If an additional mapping request of the same physical memory page arrives to the IOMMU mapping layer while a mapping already exists for that page, the old mapping is reused. If an old mapping is not used within the pre-defined time limit, it is unmapped completely and the corresponding IOMMU PTEs are invalidated, limiting the overall window of vulnerability for an errant DMA to the pre-defined time limit. We determined experimentally that on our system a modest limit of ten milliseconds is enough to achieve a 92% hit rate.

We keep track of all cached mappings in the same software LRU cache, regardless of how many times each mapping is shared. Mappings which are not currently in use are also kept in a deferred-unmappings first-in first-out (FIFO) queue with a fixed size limit. The queue size and the residency constraints are checked whenever the queue is accessed, and also periodically. Invalidations are performed when mappings are removed from the queue.

4 Sidecore IOMMU Emulation

Samecore emulation uses the classical approach of trapping device register access and switching to the hypervisor for handling. We now present an alternative, novel approach for device emulation which uses a second core to handle device register accesses, thus avoiding expensive VM-exits. We call this sidecore emulation. While the discussion below focuses on Intel's VT-d, our approach is generic and can be applied to most other IOMMUs and I/O devices.

Samecore hardware emulation suffers from an inherent limitation. Each read or write access to the hardware registers requires a VM-transition to the hypervisor, which then emulates the hardware behavior. VM-transitions are known to be expensive, partly due to cache pollution [2, 11].

Offloading computation to a sidecore for speeding up I/O for modified (paravirtualized) guests has been explored by Kumar et al. [23], Gavrilovska et al. [15], and Liu and Abali [27]. Sidecore emulation offloads device emulation to a sidecore. In contrast with previous paravirtualized sidecore approaches, which require guest modifications, sidecore emulation maintains the same hardware interface between the guest and the sidecore as between a bare-metal OS and the real hardware device, and thus requires no guest modifications. As we show in Section 6, sidecore emulation on its own can achieve 69% of bare metal performance—for unmodified guests and without any protection relaxation.

In general, hardware emulation by a sidecore follows the same principles as samecore emulation. The guest programs the device, the hypervisor detects that the guest has accessed the device, decodes the semantics of the access, and emulates the hardware behavior. But sidecore emulation differs from samecore emulation in two fundamental aspects. First, there are no expensive traps from the guest to the hypervisor when the guest accesses device registers. Instead, the device register memory areas are shared between the guest and the hypervisor, and the hypervisor polls the emulated control registers for updates. Second, the guest code and the hypervisor code run on different cores, leading to reduced cache pollution and improved utilization of each core's exclusive caches.

Efficient hardware emulation by a sidecore is dependent on the interface between the I/O device and the guest OS, since the sidecore polls memory regions instead of

receiving notifications on discrete register access events. In general, efficient sidecore emulation requires that the physical hardware have the following (commonly found) properties.

Synchronous Register Write Protocol Sidecore emulation relies on a synchronous protocol between the device driver and the device for a single register's updates, in the sense that the device driver expects some indication from the hardware before writing to a register a second time. Such a protocol ensures that the sidecore has time to process the first write before a second write to the same register overwrites the first write's contents.

A Single Register Holds Only Read-Only or Write Fields Registers which hold both read-only and write fields are challenging for a sidecore to handle, since the sidecore has no efficient way of ensuring the guest device driver would not change read-only fields.

Loose Response Time Requirements Sidecore emulation is likely to be slower than physical hardware. If the device has strict specifications of the "wall time" device operations take (e.g., "this operation completes within 3ns") or the device driver makes other strong timing assumptions which hold for real hardware but not for emulated hardware, then the device driver might assume that the hardware is malfunctioning when operations take longer than expected. This property must hold for device emulation in general.

Explicit Update of Memory-Resident Data Structures Since the sidecore cannot poll large memory regions efficiently, update to its memory-resident data structures should be explicit, by requiring the device-driver to perform a write-access to the device control registers indicating exactly which data structure it updated.

An additional, optional property that can boost sidecore emulation performance is a limited number of control registers. Since the sidecore needs to sample the control registers of the emulated hardware, a large number of registers would result in long latency between the time the guest sets the control register and the time the sidecore detects the change. In addition, polling a large number of registers may result in cache thrashing.

Intel's IOMMU has all of the properties required for efficient sidecore emulation. This is in contrast to AMD's IOMMU, which cannot require the OS's mapping layer to explicitly update the IOMMU registers upon every change to the memory-resident page tables. We note, however, that the emulated IOMMU and the platform's physical IOMMU are orthogonal, and Intel's IOMMU can be emulated when only AMD's IOMMU is physically present or even when no physical IOMMU is present and bounce buffers are used instead [12].

5 Reasoning About Risk and Protection

5.1 Risk and Protection Types

The IOMMU was designed to protect those pages which do not hold I/O buffers from errant DMA transactions. To achieve complete protection, the IOMMU mapping layer must ensure a page is accessible for DMA transactions only if it holds an I/O buffer that may be used for DMA transaction and only while a valid DMA transaction may target this page [49].

However, IOMMU mapping layer optimizations may relax protection by completing the synchronous unmap function call by the I/O device driver (*logical unmapping*) before tearing down the mapping in the physical IOMMU page-tables and completing the physical IOTLB invalidation (*physical unmapping*).

Deferring physical unmapping this way, as done by the deferred invalidation scheme, the asynchronous invalidation scheme, and the optimistic teardown scheme, could potentially compromise protection for any page which has been logically unmapped but not yet physically unmapped. We differentiate, however, between *inter-guest protection*, protection between different guest OS instances, and *intra-guest protection*, protection within a particular guest OS [44].

vIOMMU maintains full inter-guest protection—full isolation between VMs—in all configurations. It maintains inter-guest protection by keeping pages pinned in physical memory until they have been physically unmapped. vIOMMU *pins* a page in physical memory before mapping it in the IOMMU page table, and only *unpins* it once the IOMMU mapping of that page is torn down and the IOTLB invalidation is complete. Consequently, any page that is used for a DMA transaction by a guest OS will not be re-allocated to any other guest OS as long as it may be the target of a valid DMA transaction by the first guest OS.

Full intra-guest protection—protecting a guest OS from itself—is arguably less important than inter-guest protection in a virtualized setting. Intra-guest protection may be relaxed by both the host's and the guest's mapping layer optimizations. Maintaining complete intra-guest protection with optimal performance in an operating system such as Linux without modifying all drivers remains an open challenge [49], since Linux drivers assume that any page that has been logically unmapped is also physically unmapped. Consequently, such pages are often re-used by the driver or the I/O stack for other purposes as soon as they have been logically unmapped.

5.2 Quantifying Risk

We do not assess the risk posed by arbitrary malicious adversaries, since such adversaries might sometimes be able exploit even very short vulnerability windows [40]. In our discussion of protection and risk we focus instead on the "window of vulnerability", when an errant DMA may sneak in and read or write an exposed I/O buffer through a *stale mapping*. A stale mapping is a mapping which exists after a page has been logically unmapped but before it has been physically unmapped. A stale mapping occurs when the device driver asks to unmap an IOVA translation and receives an affirmative response, despite the actual teardown of the physical IOMMU PTE or physical IOTLB invalidation having been deferred.

We quantify risk along two axes: the *duration of vulnerability* during which an I/O buffer is open for reading or writing through a stale mapping, and the *stale mapping bound*, which indicates the maximum number of stale mappings at any given point in time.

We classify the mapping strategies mentioned above into four classes according to their duration: no risk, nanosecond risk, microsecond risk, and millisecond risk.

No Risk The only times when there is no risk are when an OS on bare metal uses the strict mapping strategy, or when both guest and host use the strict mapping strategy. Since buffers are unmapped and their mappings invalidated without any delay, there can be no stale mappings regardless of whether we run on bare metal, use samecore emulation, or sidecore emulation. The use of approximate shared mappings does not affect the risk.

Nanosecond Risk The time that elapses between the moment when the host posts an invalidation request to the invalidation queue and the invalid translation is actually flushed from the physical IOTLB can be measured in nanoseconds. Since the flush happens in silicon, this duration is a physical property of the platform IOMMU, and the risk only applies to bare metal with asynchronous invalidation. With samecore or sidecore emulation, guest/host communication costs overshadow this duration. We determined experimentally that on our system the stale mapping bound for nanosecond risk is at most two mappings, and the duration of vulnerability is 128 cycles per entry on average.

Microsecond Risk Microsecond risk only applies to sidecore emulation and comes into play when the guest does not wait for the host to process an invalidation (i.e., when the guest uses asynchronous invalidation). Here, inter-core communication costs determine the window of vulnerability, since the host must realize that the guest posted an invalidation before it can handle it. In general, the stale mapping bound for microsecond risk is the number of outstanding invalidation requests in the emulated invalidation queue. In our experimental setup the queue was sized to hold at most 128 outstanding entries.

Millisecond Risk Millisecond risk applies when either the guest or the host uses the deferred invalidation or optimistic teardown strategies. Regardless of whether the guest or the host defers invalidations or keeps around cached mappings, the window of vulnerability is likely to be in the order of milliseconds. Software configures the stale mappings bounds by setting a quota on the number of cached mappings and a residency time limit on each mapping.

Overall Risk When a guest OS uses an emulated IOMMU, the combination of the guest's and host's mapping strategies determines the overall protection level. The hypervisor cannot override the guest mapping strategy to provide greater protection, since the hypervisor is unaware of any cached mappings or deferred invalidations in the guest until the guest unmaps them and executes the invalidations. Therefore, the hypervisor can either keep the guest's level of protection by using a strict invalidation scheme, or relax it for better performance.

6 Performance Evaluation

6.1 Methodology

Experimental Setup We implement the samecore and sidecore emulation of Intel IOMMU, as well as the mapping layer optimizations presented above. We use the KVM hypervisor [21] and Ubuntu 9.10 running Linux 2.6.35 for both host and guest. Our experimental setup is comprised of an IBM System x3550 M2, which is a dual-socket, four-cores per socket server equipped with Intel Xeon X5570 CPUs running at 2.93GHz. The Chipset is Intel 5520, which supports VT-d. The system includes 16GB of memory and an Emulex OneConnect 10Gbps NIC. We use another identical remote server (connected directly by 10Gbps optical fiber) as a workload generator and a target for I/O transactions. In order to obtain consistent results and to avoid reporting artifacts caused by nondeterministic events, all power optimizations are turned off, namely, sleep states (C-states) and DVFS (dynamic voltage and frequency scaling).

To have comparable setups, guest-mode configurations execute with a single VCPU (virtual CPU), and native-mode configurations likewise execute with a single core enabled. In guest-mode setups, the VCPU and the sidecore are pinned to two different cores on the same die, and 2GB of memory is allocated to the guest.

Microbenchmarks We use two well-known Netperf [19] instances in order to assess the overheads induced by vIOMMU in terms of throughput, CPU cycles,

config.	*guest/native invalidation*	*guest/native reuse*	*guest/native Linux*	*host invalidation*	*native max stale#*	*guest max stale#*	*duration magnitude*
strict	strict	none	unpatched	strict	none	none	0
shared	strict	shared	patched	strict	none	none	0
async	async	shared	patched	async	32	128+32	μsec
deferred	deferred	none	unpatched	deferred	32	250+32	ms
opt256	async	shared+tear	patched	deferred	256+32	256+128+32	ms
opt4096	async	shared+tear	patched	deferred	4096+32	4096+128+32	ms
off	n/a	n/a	unpatched	deferred	all	all	∞

Table 2: *Evaluated configurations. The host column is meaningless when running the native configuration. The maximal number of stale mappings for async and deferred host is the size of the IOTLB, namely, 32.*

and latency. The first instance—Netperf TCP stream—attempts to maximize the amount of data sent over a single TCP connection, simulating an I/O-intensive workload. The second instance–Netperf UDP RR (request-response)—models a latency-sensitive workload by repeatedly sending a single byte and waiting for a matching single byte response. Latency is calculated as the inverse of the number of transactions per second.

Macrobenchmarks We use two macrobenchmarks to asses the performance of vIOMMU on real applications. The first is MySQL SysBench OLTP (version 0.4.12; executed with MySQL database version 5.1.37), which was created for benchmarking the MySQL database server by generating OLTP inspired workloads. To simulate high-performance storage, the database is placed on a remote machine's RAM-drive, which is accessed through NFS and mounted in synchronous mode. The database contains two million records, which collectively require about 1GB. We disable data caching on the server by using the InnoDB engine and the `O_DIRECT` flush method.

The second macrobenchmark we use is Apache Bench, evaluating the performance of the Apache web server. Apache Bench is a workload generator that is distributed with Apache to help assess the number of concurrent requests per second that the server is capable of handling. The benchmark is executed with 25 concurrent requests. The logging is disabled to avoid the overhead of writing to disk.

Configurations There are many possible combinations of emulation approaches, which are comprised of the guest and host mapping layers and their reuse and invalidation strategies. Each such combination is associated with different protection and performance levels. We cannot evaluate all combinations. We instead choose to present several meaningful ones in the hope that they provide reasonable coverage. The configurations are listed in Table 2. Each line in the table pertains to two scenarios: a virtualized setting, with a guest serviced by a host, and a "native" setting with only the bare metal OS running. The latter scenario provides a baseline. It is addressed because our optimizations apply to virtualized settings and bare metal settings alike. We next describe the configurations one by one, from safest to riskiest.

The **strict** configuration involves no optimizations in guest, host, or native modes, and hence it involves no risk; it is the least performant configuration. While strict is not the default mode of Linux, it requires no OS modification, but rather setting an already-existing configurable parameter. Hence it is marked as "unpatched".

The **shared** configuration is nearly identical to strict except that it adds the approximate shared mapping optimization (Section 3.1); it is still risk-free, merely attempting to avoid allocating more than one IOVA for a given physical location and preferring instead to reuse. Notice that for the virtualized setting this optimization is meaningless for the host, as the hypervisor cannot override the IOVA chosen by the guest. The OS is patched because Linux does not natively support shared mappings.

The **async** configuration is similar to the shared configuration, yet in addition it utilizes the asynchronous IOTLB invalidation optimization (Section 3.2). The latter immediately invalidates unmapped translations, but does not wait for the IOTLB invalidation to complete, reducing invalidation cost by the time it takes the IOMMU to write its invalidation completion message back to memory. Realistically, the risk exists only for the sidecore setting, which is dominated by inter-core communication cost that is approximated by not more than a handful of μsecs. The theoretical maximal number of stale entries is the size of the IOTLB (32) in the host and native settings; in this guest's case, this is supplemented by the default size of the invalidation queue (128).

The **deferred** configuration is the default configuration of Linux, whereby IOTLB invalidations are aggregated and processed together every 10ms (Section 3.3). In the guest's case, stale entries might reside in the IOTLB (32) or in the deferred entries queue (up to 250 by

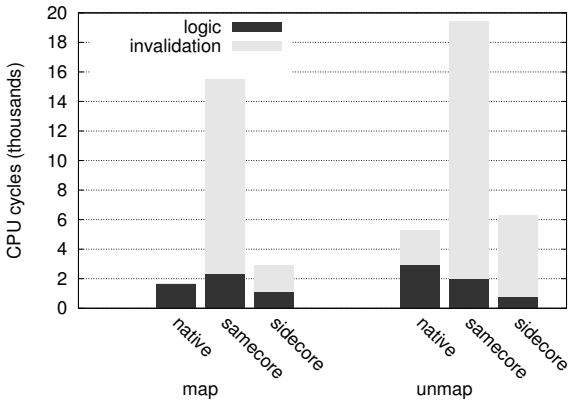


Figure 2: *Average breakdown of (un)mapping a single page using the strict invalidation scheme.*

default). While the entries are in the guest's queue, the host does not know about them and hence cannot invalidate them. As both guest and host use a 10ms interval, the per-entry maximal vulnerability window is 20ms for the guest and half that much for the host and bare metal.

The **opt** configuration (short for "optimistic") deploys all optimizations save deferred invalidation, which is substituted by optimistic teardown (Section 3.4). The maximal number of stale entries we keep alive (for up to 10ms) is 256, similarly to the 250 of deferred; in a more aggressive configuration we increase that number to 4096.

Finally, the **off** configuration does not employ an IOMMU in the native setting, and does not employ a vIOMMU in the virtualized setting. (In the latter case the physical IOMMU is nevertheless utilized by the host, because the device is still assigned to the guest.) In this configuration, neither the guest nor the native bare metal enjoy any form of protection, which is why we marked "all" the mappings as unsafe for their entire lifetime ("∞").

6.2 Overhead of (Un)mapping

The IOMMU layer provides exactly two primitives: map and unmap. Before we delve into the benchmark results, we first profile the overhead induced by the vIOMMU with respect to these two operations. Figure 2 presents the cycles breakdown of each operation to IOTLB "invalidation", which is the direct interactions of the OS with the IOMMU, and to "logic", which encapsulates the rest of the code that builds and destroys the mappings within the I/O page tables.

Notice that guest invalidation overhead is induced even when performing the map operation; this happens because the hypervisor turns on the "caching mode bit", which, by the IOMMU specification, means that the OS is mandated to first invalidate every new mapping it creates (which allows the hypervisor to track this activity). Most evident in the figure is the fact that the sidecore dramatically cuts down the price of invalidation when compared to samecore, which is a direct result of eliminating the associated VM exits and associated world switches. The other interesting observation is that the rest of the (un)map logic can be accomplished faster by the vIOMMU. This better-than-native performance is a product of the vIOMMU registers being cacheable, as opposed to those of the physical IOMMU.

6.3 Benchmark Results

Figure 3(a) depicts the throughput of Netperf/TCP for each configuration, from safest to riskiest, along the X axis. The values displayed are normalized by the maximal throughput achieved by bare metal and off, which in this case is 100% of the attainable bandwidth of the 10Gbps NIC. Figure 3(b) presents the very same data, but the normalization is done against native on a per-configuration basis; accordingly, the native curve coincides with the "1" grid line. Figure 3(c) presents the CPU consumed by Netperf/TCP while doing the corresponding work; observe that the sidecore is associated with two curves in this figure, the lower one corresponds to the useful work done by the sidecore (aside from polling) and the upper one pertains to the main core.

The safe (shared) or nearly safe (async) configurations provide no benefit for the samecore setting, but they can slightly improve the performance of sidecore and native by 2–5 percentage points each. Deferred delivers a much more pronounced improvement, especially in the native case, which manages to attain 91% of the line-rate. By consulting Figure 3(c), we can see that native/deferred is not attaining 100%, because the CPU is a bottleneck. Utilizing opt solves this problem, not only for the native setting, but also for the sidecore; opt allows both to fully exploit the NIC. The sidecore/CPU curve (bottom of Figure 3(c)) implies that the work required form the IOMMU software layer is little when optimal teardown is employed, allowing the sidecore to catch up with native performance and the samecore to reduce to gap to 0.82x the optimum.

Similarly to the above, Figure 4 depicts the latency as measured with Netperf/UDP-RR and the associated CPU consumption. The results largely agree with what we have seen for Netperf/TCP. Deferring the IOTLB invalidation allows the native setting to achieve optimal latency, but only slightly improves the virtualized settings. However, when optimistic teardown is employed, the latency of both sidecore and samecore drops significantly (by about 60 percentage point in the latter case), and they manage attain the optimum. Importantly, the

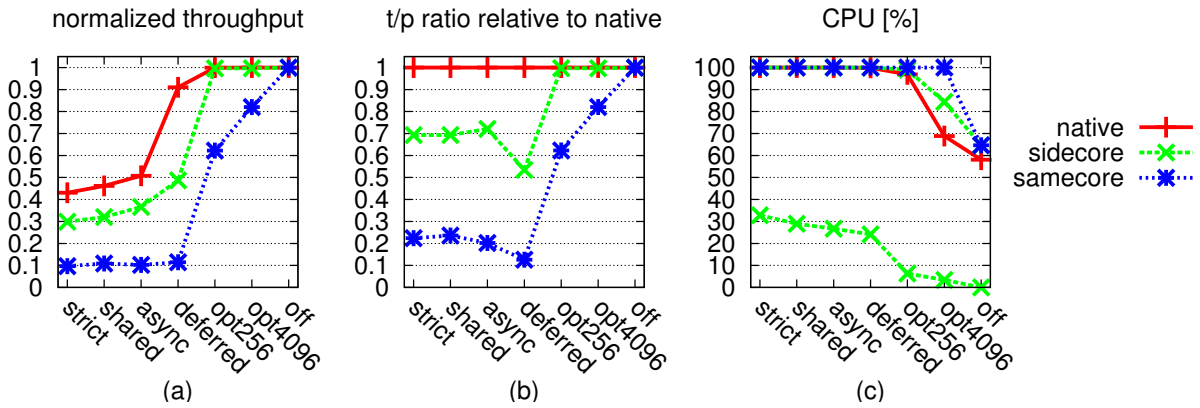


Figure 3: *Measuring throughput with Netperf TCP; the baseline for normalization is the optimal throughput attainable by our 10Gbps NIC.*

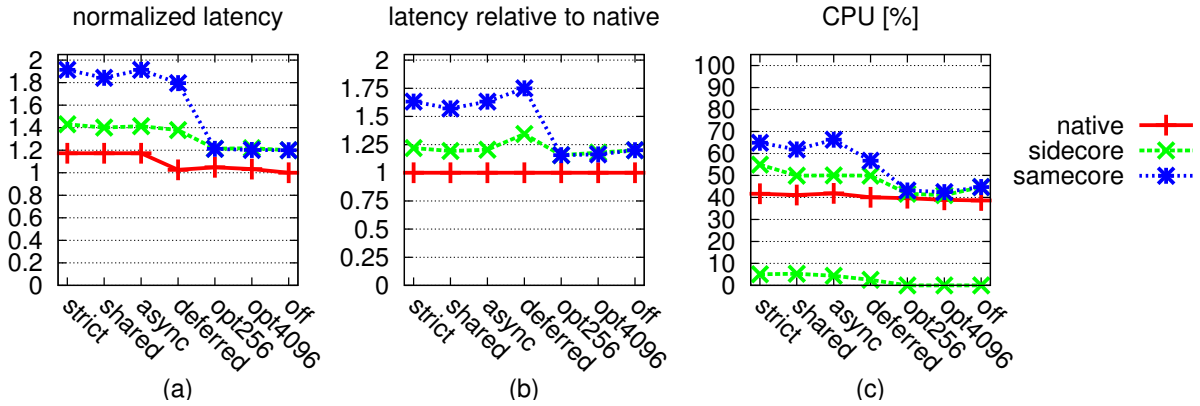


Figure 4: *Measuring latency with the Netperf UDP request-response benchmark; the baseline for normalization (latency of bare metal with no IOMMU) is 41 μsecs.*

optimum for the samecore and sidecore settings is not the "1" that is shown in Figure 4(a); rather, it is the value that is associated with the off configuration of the virtualized settings (guest with no IOMMU protection), which is roughly 1.2 in this case.

Examining Figure 4(c), we unsurprisingly see that the CPU is not a bottleneck for this benchmark. We further see that optimistic teardown is the most significant optimization for this metric, allowing the virtualized settings to nearly reach the bare metal optimum.

Figures 5–6 present the results of the macrobenchmarks, showing trends that are rather similar. Optimistic teardown is most meaningful to the samecore setting, boosting its throughput by about 1.5x. For sidecore, however, the optimization has a lesser effect. Specifically, opt4096 improves upon deferred by 1.07x in the case of MySQL, and by 1.04x in the case of Apache. Before the optimistic teardown is applied, the sidecore setting delivers 1.52x and 1.63x better throughput than samecore for MySQL and Apache, respectively. But once it is applied, then these figures respectively drop to 1.12x and 1.10x. In other words, for the real applications that we have chosen, sidecore is better than samecore by 50%–60% for safe configurations (as well as for deferred), but when optimistic teardown is applied, this gap is reduced to around 10%. This should come as no surprise as we have already established above that optimistic teardown dramatically reduces the IOMMU overhead.

It is important to note, once again, that the optimum for sidecore and samecore is the off configuration in the virtualized setting, namely 0.86 and 0.68 for MySQL and Apache in Figures 5(a) and 6(a), respectively. Thus, it is not that the optimistic teardown all of a sudden became less effective for the macrobenchmarks; rather, it is that in comparison to the microbenchmarks the applications attain much higher throughput to begin with, and so the optimization has less room to shine.

The bottom line is that combining sidecore and op-

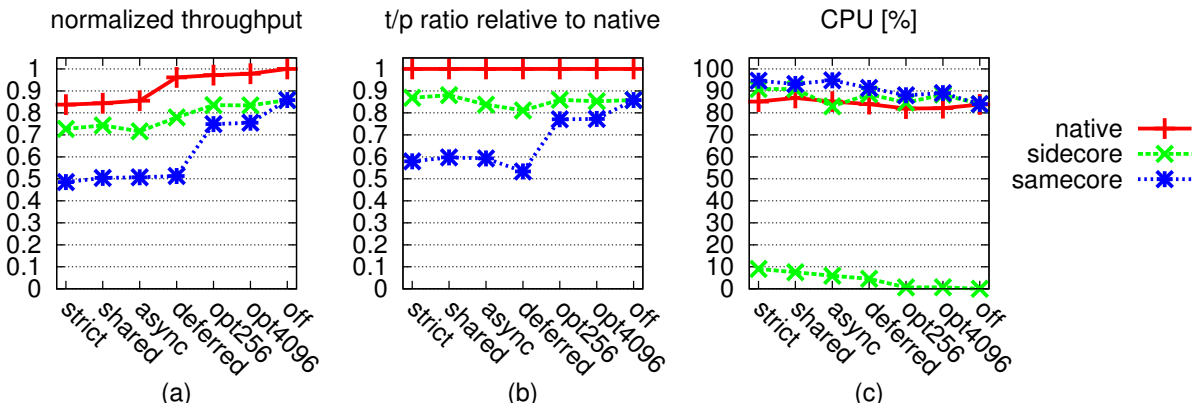


Figure 5: *Measuring MySQL throughput; the baseline for normalization is 243 transactions per second.*

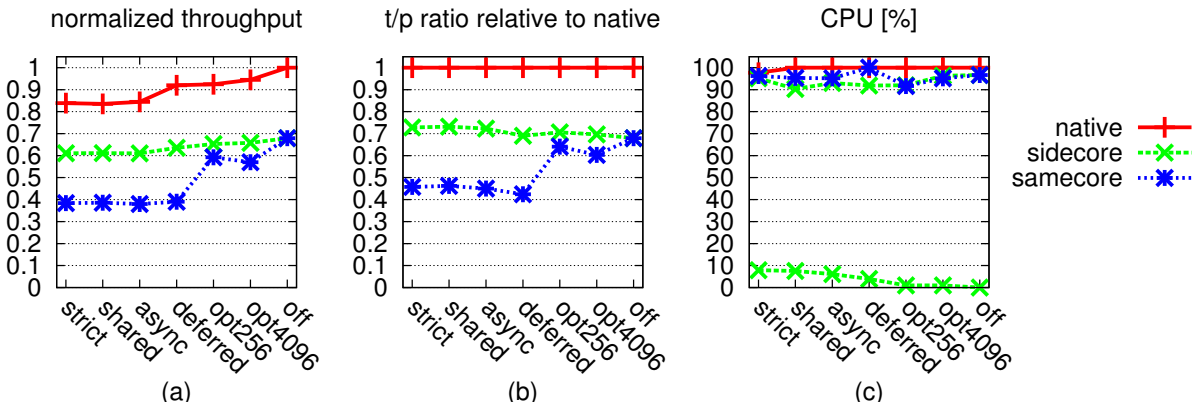


Figure 6: *Measuring Apache throughput; the baseline for normalization is 6828 requests per second.*

	Throughput(Mbps)	*VCPUs load*	*Sidecore load*
samecore	1345 (+49%)	76%	
sidecore	4312 (+54%)	83%	49% (+50%)

Table 3: *Measuring the Netperf TCP throughput of 2-VCPUs with strict configuration compared to a single VCPU.*

timistic teardown brings both MySQL and Apache throughputs to be only 3% less than their respective optima.

6.4 Sidecore Scalability and Power-Efficiency

Performance gain from the sidecore approach requires the emulating sidecore to be co-scheduled with the VCPUs to achieve low-latency IOMMU emulation. Therefore, it is highly important that the sidecore performs its tasks efficiently with high utilization.

One method for better utilizing the sidecore is to set one emulating sidecore to serve multiple VCPUs or multiple guest CPUs. Table 3 presents the performance of a 2 VCPUs setup, using the strict configuration, relative to a single VCPU setup. As shown, sidecore emulation scales up similarly to samecore emulation, and the performance of both improves by approximately 50% in 2 VCPUs setup.

This method, however, may encounter additional latency in a system that consists multiple sockets (dies), as the affinity of the sidecore thread has special importance in such systems. If both the virtual guest and the sidecore are located on the same die, fast cache-to-cache micro-architectural mechanisms can be used to propagate modifications of the IOMMU data structures, and the interconnect imposes no additional latency. In contrast, when the sidecore is located on a different die, the latency of accessing the emulated IOMMU data structures is increased by interconnect imposed latency. The Intel QuickPath Interconnect (QPI) protocol used on our system requires write-backs of modified cache lines to main memory, which results in latency that can exceed 100ns—over four times the latency of accessing a modi-

fied cache line on the same die [30].

Another method for better utilizing the sidecore is to use its spare cycles productively. Even though the nature of the sidecore is that it is constantly working, a sidecore can have spare cycles—those cycles in which it polled memory and realized it has no pending emulation tasks. One way of improving the system's overall efficiency is to use such cycles for polling paravirtual devices in addition to emulated devices. Another way is to allow the sidecore to enter a low-power sleep state when it is otherwise idle. We can make sidecore IOMMU emulation more power-efficient by using the CPU's monitor/mwait capability, which enables the core to enter a low-power state until a monitored cache range is modified [4].

However, current x86 architecture only enables monitoring of a single cache line, and the Linux scheduler already uses the monitoring hardware for its internal purposes. Moreover, the sidecore must monitor and respond to writes to multiple emulated registers which do not reside in the same cache line.

We overcame these challenges by using the mapping hardware to monitor the *invalidation queue tail* (IQT) register of the IOMMU invalidation queue while we periodically monitored the remaining emulated IOMMU registers. (This is possible because the IOMMU mapping layer performs most of its writes to a certain IQT register.) We also relocated the memory range monitored by the scheduler (the `need_resched` variable) to a memory area which is reserved according to the IOMMU specifications and resides in the same cache line as the IQT register.

Nonetheless, entering a low-power sleep state is suitable only in an extended quiescence period, in which no accesses to the IOMMU take place. This is because entering and exiting low power state takes considerable time [39]. Thus, sidecore emulation is ideally suited for an asymmetric system [22]. Such systems, which include both high power high performance cores and low power low performance cores, can schedule the hardware emulation code to a core which will provide the desired performance/power consumption tradeoff.

The impact of these two scaling related methods, using sidecore to serve a guest whose VCPU is located on another package, and entering low power state instead of polling, appear in Figure 7. According to our experiments, when the sidecore was set on another package, the mapping and unmapping cost increased by 23%, resulting in 25% less TCP throughput than when the sidecore was located on the same package. Entering low-power state increased the cycle cost of mapping and unmapping by 13%, and optimally would decrease performance very little using good heuristics for detecting idle periods. Regardless, in both cases, the cost of sidecore emulation is still considerably lower than that of samecore emulation.

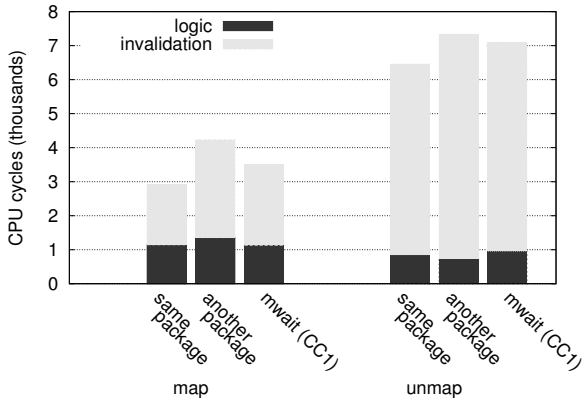


Figure 7: *The effect of power-saving and CPU affinity on the mapping/unmapping cost of a single page.*

7 Related Work

We survey related work along the following dimensions: I/O device emulation for virtual machines, IOMMU mapping strategies for paravirtualized and unmodified guests, and offloading computation to a sidecore.

All common hypervisors in use today on x86 systems emulate I/O devices. Sugerman, Venkitachalam, and Lim discuss device emulation in the context of VMware's hypervisor [37], Barham et al. discuss it in the context of the Xen hypervisor [6], Kivity et al. discuss it in the context of the KVM hypervisor [21], and Bellard discusses it in the context of QEMU [10]. In all cases, device emulation suffered from prohibitive performance [11], which led to the development of paravirtualized I/O [6, 34] and direct device assignment I/O [25, 26]. To our knowledge, we are the first to demonstrate the feasibility of high-speed I/O device emulation with performance approaching that of bare metal.

Maximizing OS protection from errant DMAs by minimizing the DMA vulnerability duration is important, because devices might be buggy or exploited [9, 14, 25, 46]. Several IOMMU mapping strategies have been suggested for trading off protection and performance [44, 49]. For unmodified guests, the only usable mapping strategy prior to this work was the direct mapping strategy [44], which provides no protection to the guest OS. Once we expose an emulated IOMMU to the guest OS, the guest OS may choose to use any mapping strategy it wishes to protect itself from buggy or malicious devices.

Additional mapping strategies were possible for paravirtualized guests. The single-use mapping and the shared mapping strategies provide full protection at sizable cost to performance [44]. The persistent mappings strategy provides better performance at the expense of reduced protection. In the persistent mapping strategy

mappings persist forever. The on-demand mapping strategy [49] refines persistent mapping by tearing down mappings once a set quota on the number of mappings was reached. On-demand mapping, however, does not limit the duration of vulnerability. Optimistic teardown provides performance that is equivalent to that of persistent and on-demand mapping, but does so while limiting the duration of vulnerability to mere milliseconds.

Offloading computation to a dedicated core is a well-known approach for speeding up computation [7, 8, 36]. Offloading computation to a sidecore in order to speed up I/O for paravirtualized guests was explored by Kumar et al. [23], Gavrilovska et al. [15], and in the virtualization polling engine (VPE) [27]. In order to achieve near native performance for 10GbE, VPE required modifications of the guest OS and a set of paravirtualized drivers for each emulated device. In contrast, our sidecore emulation approach requires no changes to the guest OS.

Building in part upon vIOMMU, the SplitX project [24] takes the sidecore approach one step further. SplitX aims to run each unmodified guest and the hypervisor on a disjoint set of cores, dedicating a set of cores to each guest and offloading all hypervisor functionality to a disjoint set of sidecores.

8 Conclusions

We presented vIOMMU, the first x86 IOMMU emulation for unmodified guests. By exposing an IOMMU to the guest we enable the guest to protect itself from buggy device drivers, while simultaneously making it possible for the hypervisor to overcommit memory. vIOMMU employs two novel optimizations to perform well. The first, "optimistic teardown", entails simply waiting a few milliseconds before tearing down an IOMMU mapping and demonstrates that a minuscule relaxation of protection can lead to large performance benefits. The second, running IOMMU emulation on a sidecore, demonstrates that given the right software/hardware interface and device emulation, unmodified guests can perform just as well as paravirtualized guests.

The benefits of IOMMU emulation rely on the guest using the IOMMU. Introducing software and hardware support for I/O page faults could relax this requirement and enable seamless memory overcommitment even for non-cooperative guests. Likewise, introducing software and hardware support for multiple levels of IOMMU page tables [11] could in theory provide perfect protection without any decrease in performance. In practice, multiple MMU levels cause more page-faults and higher TLB miss-rates, resulting in lower performance for many workloads [42]. Similarly, a single level of IOMMU emulation may perform better than multiple levels of IOMMU page tables, depending on workload.

Acknowledgments

We thank Ben-Ami Yassour, Abel Gordon, Nadav Har'El, and Alex Landau for their insightful comments and joyful discussions. We also thank the anonymous reviewers and Carl Waldspurger (our shepherd) for their much appreciated feedback. The research leading to the results presented in this paper is partially supported by the European Community's Seventh Framework Programme ([FP7/2001-2013]) under grant agreement numbers 248615 (IOLanes) and 248647 (ENCORE).

References

[1] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. *Intel Technology Journal*, 10(3):179–192, Aug 2006.

[2] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ACM Archit. Support for Prog. Lang. & Operating Syst. (ASPLOS)*, pages 2–13, 2006.

[3] N. Amit, M. Ben-Yehuda, and B.-A. Yassour. IOMMU: Strategies for mitigating the IOTLB bottleneck. In *Workshop on Interaction between Opearting Syst. & Comput. Archit. (WIOSCA)*, 2010.

[4] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pages 1–8, 2008.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *ACM SIGOPS European Conf. on Comput. Syst. (EuroSys)*, pages 75–85, 2006.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 164–177, 2003.

[7] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 29–44, 2009.

[8] A. Baumann, S. Peter, A. Schüpbach, A. Singhania, T. Roscoe, P. Barham, and R. Isaacs. Your computer is already a distributed system. Why isn't your OS? In *USENIX Workshop on Hot Topics in Operating Syst. (HOTOS)*, page 12, 2009.

[9] M. Becher, M. Dornseif, and C. N. Klein. FireWire: all your memory are belong to us. In *CanSecWest*, 2005.

[10] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Ann. Technical Conf. (ATC)*, pages 41–46, 2005.

[11] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The turtles project: Design and implementation of nested virtualization. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 423–436, 2010.

[12] M. Ben-Yehuda, J. Mason, O. Krieger, J. Xenidis, L. Van Doorn, A. Mallick, J. Nakajima, and E. Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Ottawa Linux Symp. (OLS)*, pages 71–86, 2006.

[13] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symp. (OLS)*, pages 9–20, 2007.

[14] B. D. Carrier and J. Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[15] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya. High-performance hypervisor architectures: Virtualization in HPC systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*, 2007.

[16] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Comm. of the ACM (CACM)*, 53(10):85–93, Oct 2010.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *IEEE Int'l Conf. on Dependable Syst. & Networks (DSN)*, pages 41–50, 2007.

[18] Intel virtualization technology for directed I/O, architecture specification. `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`, Feb 2011. Revision 1.3. Intel Corporation. (Accessed Apr 2011).

[19] R. Jones. The netperf benchmark. `http://www.netperf.org`. (Accessed Apr, 2011).

[20] A. Kadav and M. M. Swift. Live migration of direct-access devices. In *USENIX Workshop on I/O Virtualization (WIOV)*, page 2, 2008.

[21] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Ottawa Linux Symp. (OLS)*, pages 225–230, 2007. `http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf`. (Accessed Apr, 2011).

[22] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *ACM/IEEE Int'l Symp. on Comput. Archit. (ISCA)*, pages 81–92, 2004.

[23] S. Kumar, H. Raj, K. Schwan, and I. Ganev. Re-architecting VMMs for multicore systems: The sidecore approach. In *Workshop on Interaction between Opearting Syst. & Comput. Archit. (WIOSCA)*, 2007.

[24] A. Landau, M. Ben-Yehuda, and A. Gordon. SplitX: Split guest/hypervisor execution on multi-core. In *USENIX Workshop on I/O Virtualization (WIOV)*, 2011.

[25] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 17–30, 2004.

[26] J. Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support. In *IEEE Int'l Parallel & Distributed Processing Symp. (IPDPS)*, pages 1–12, 2010.

[27] J. Liu and B. Abali. Virtualization polling engine (VPE): Using dedicated CPU cores to accelerate I/O virtualization. In *ACM Int'l Conf. on Supercomputing (ICS)*, pages 225–234, 2009.

[28] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance VMM-bypass I/O in virtual machines. In *USENIX Ann. Technical Conf. (ATC)*, pages 29–42, 2006.

[29] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 13–23, 2005.

[30] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *ACM/IEEE Int'l Conf. on Parallel Archit. & Compilation Techniques (PACT)*, pages 261–270, 2009.

[31] E. G. Munteanu. AMD IOMMU emulation patchset. KVM mailing list, `http://www.spinics.net/lists/kvm/msg38514.html`, Jul 2010. (Accessed Apr, 2011).

[32] B. Pfaff. Performance analysis of BSTs in system software. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 32(1):410–411, Jun 2004.

[33] H. Raj and K. Schwan. High performance and scalable I/O virtualization via self-virtualized devices. In *Int'l Symp. on High Performance Distributed Comput. (HPDC)*, pages 179–188, 2007.

[34] R. Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Syst. Review (OSR)*, 42(5):95–103, Jul 2008.

[35] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *ACM Symp. on Operating Syst. Principles (SOSP)*, pages 335–350, 2007.

[36] L. Shalev, J. Satran, E. Borovik, and M. Ben-Yehuda. IsoStack: highly efficient network processing on dedicated cores. In *USENIX Ann. Technical Conf. (ATC)*, page 5, 2010.

[37] J. Sugerman, G. Venkitachalam, and B.-H. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Ann. Technical Conf. (ATC)*, pages 1–14, 2001.

[38] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst. (TOCS)*, 23(1):77–110, Feb 2005.

[39] D. Tsafrir, Y. Etsion, and D. G. Feitelson. General purpose timing: the failure of periodic timers. Technical Report 2005-6, School of Computer Science & Engineering, the Hebrew University, Feb 2005.

[40] D. Tsafrir, T. Hertz, D. Wagner, and D. Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *USENIX Conf. on File & Storage Technologies (FAST)*, pages 189–206, 2008.

[41] C. A. Waldspurger. Memory resource management in VMware ESX server. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, volume 36, pages 181–194, 2002.

[42] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. Selective hardware/software memory virtualization. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 217–226, 2011.

[43] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *USENIX Symp. on Opearting Syst. Design & Implementation (OSDI)*, pages 241–254, 2008.

[44] P. Willmann, S. Rixner, and A. L. Cox. Protection strategies for direct access to virtualized I/O devices. In *USENIX Ann. Technical Conf. (ATC)*, pages 15–28, 2008.

[45] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *IEEE Int'l Symp. on High Performance Comput. Archit. (HPCA)*, pages 306–317, 2007.

[46] R. Wojtczuk. Subverting the Xen hypervisor. In *Black Hat*, 2008. `http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf`. (Accessed Apr, 2011).

[47] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner. Memory buddies: exploiting page sharing for smart colocation in virtualized data centers. In *ACM/USENIX Int'l Conf. on Virtual Execution Environments (VEE)*, pages 31–40, 2009.

[48] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. Direct device assignment for untrusted fully-virtualized virtual machines. Technical Report H-0263, IBM Research, 2008.

[49] B.-A. Yassour, M. Ben-Yehuda, and O. Wasserman. On the DMA mapping problem in direct device assignment. In *Haifa Experimental Syst. Conf. (SYSTOR)*, 2010.

[50] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for Linux VM. In *Ottawa Linux Symp. (OLS)*, pages 261–268, 2008.

HiTune: Dataflow-Based Performance Analysis for Big Data Cloud

Jinquan Dai, Jie Huang, Shengsheng Huang, Bo Huang, Yan Liu
Intel Asia-Pacific Research and Development Ltd
Shanghai, P.R.China, 200241
{jason.dai, jie.huang, shengsheng.huang, bo.huang, yan.b.liu}@intel.com

Abstract

Although Big Data Cloud (e.g., MapReduce, Hadoop and Dryad) makes it easy to develop and run highly scalable applications, efficient provisioning and fine-tuning of these massively distributed systems remain a major challenge. In this paper, we describe a general approach to help address this challenge, based on distributed instrumentations and dataflow-driven performance analysis. Based on this approach, we have implemented HiTune, a scalable, lightweight and extensible performance analyzer for Hadoop. We report our experience on how HiTune helps users to efficiently conduct Hadoop performance analysis and tuning, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling).

1. Introduction

There are dramatic differences between delivering software as a service in the cloud for millions to use, versus distributing software as bits for millions to run on their PCs. First and foremost, services must be highly scalable, storing and processing an enormous amount of data. For instance, in June 2010, Facebook reported 21PB raw storage capacity in their internal data warehouse, with 12TB compressed new data added every day and 800TB compressed data scanned daily [1]. This type of "Big Data" phenomenon has led to the emergence of several new cloud infrastructures (e.g., MapReduce [2], Hadoop [2], Dryad [4], Pig [5] and Hive [6]), characterized by the ability to scale to thousands of nodes, fault tolerance and relaxed consistency. In these systems, the users can develop their applications according to a dataflow graph (either implicitly dictated by the programming/query model or explicitly specified by the users). Once an application is cast into the system, the cloud runtime is responsible for dynamically mapping the logical dataflow graph to the underlying cluster for distributed executions.

With these Big Data cloud infrastructures, the users are required to exploit the inherent data parallelism exposed by the dataflow graph when developing the applications; on the other hand, they are abstracted away from the messy details of data partitioning, task distribution, load balancing, fault tolerance and node communications. Unfortunately, this abstraction makes it very difficult, if not impossible, for the users to understand the cloud runtime behaviors. Consequently, although Big Data Cloud makes it easy to develop and run highly scalable applications, efficient provisioning and fine-tuning of these massively distributed systems remain a major challenge. To help address this challenge, we attempt to design tools that allow users to understand the runtime behaviors of Big Data Cloud, so that they can make educated decisions regarding how to improve the efficiency of these massively distributed systems – just as what traditional performance analyzers do for a single execution of a single program.

Unfortunately, performance analysis for Big Data Cloud is particularly challenging, because these applications can potentially comprise several thousands of programs running on thousands of machines, and the low level performance details are hidden from the users by using a high level dataflow model. In this paper, we describe a specific solution to this problem based on distributed instrumentations and dataflow-driven performance analysis, which correlates concurrent performance activities across different programs and machines, reconstructs the dataflow-based, distributed execution process of the Big Data application, and relates the low level performance activities to the high level dataflow model.

Based on this approach, we have implemented *HiTune*, a scalable, lightweight and extensible performance analyzer for Hadoop. We report our experience on how HiTune helps users to efficiently conduct Hadoop performance analysis and tuning, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling). For instance, reconstructing the dataflow execution process of a Hadoop job allows users to understand the dynamic interactions between different tasks and stages (e.g., task scheduling and data shuffle; see sections 7.1 and 7.2). In addition, relating performance activities to the dataflow model allows users to conduct fine-grained,

dataflow-based hotspot breakdown (e.g., for identifying application hotspots and hardware problems; see sections 7.2 and 7.3).

The rest of the paper is organized as follows. In section 2, we introduce the motivations and objectives of our work. We give an overview of our approach in section 3, and present the dataflow-based performance analysis in section 4. In section 5, we describe the implementation of HiTune, a performance analyzer for Hadoop. We experimentally evaluate HiTune in section 6, and report our experience in section 7. We discuss the related work in section 8, and finally conclude the paper in section 9.

2. Problem Statement

In this section, we describe the motivations, challenges, goals and non-goals of our work.

2.1 Big Data Cloud

In Big Data Cloud, the input applications are modeled as directed acyclic dataflow graphs to the users, where graph vertices represent processing stages and graph edges represent communication channels. All the data parallelisms of the computation and the data dependencies between processing stages are explicitly encoded in the dataflow graph. The users can develop their applications by simply supplying programs that run on the vertices to these systems; on the other hand, they are abstracted away from the low level details of the distributed executions of their applications. The cloud runtime is responsible for dynamically mapping the logical dataflow graph to the underlying cluster, including generating the optimized dataflow graph of execution plans, assigning the vertices and edges to physical resources, scheduling and executing each vertex (usually using multiple instances and possibly multiple times due to failures).

For instance, the MapReduce model dictates a two-stage group-by-aggregation dataflow graph to the users, as shown in Figure 1. A MapReduce application has one input that can be trivially partitioned. In the first stage a *Map* function, which specifies how the grouping is performed, is applied to each partition of input data. In the second stage a *Reduce* function, which performs the aggregation, is applied to each group produced by the first stage. The MapReduce framework is then responsible for mapping this logical dataflow graph to the physical resources. For instance, the Hadoop framework automatically executes the input MapReduce application using an internal dataflow graph of execution plan, as shown in Figure 2. The input data is divided into *splits*, and a distinct Map task is launched for each split. Inside each Map task, the map stage applies the *Map* function to the input data, and the spill stage stores the map output on local disks. In addition, a distinct Reduce task is launched for each partition of the map outputs. Inside each Reduce task, the copier and merge stages run in a pipelined fashion, fetching the relevant partition over the network and merging the fetched data respectively; after that, the sort and reduce stages merge the reduce inputs and apply the *Reduce* function respectively.

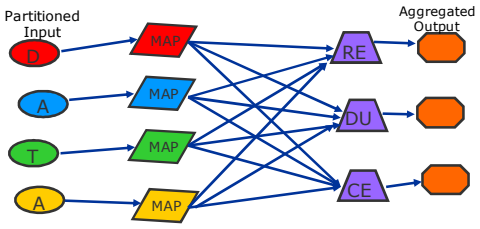


Figure 1. Dataflow graph of a MapReduce application

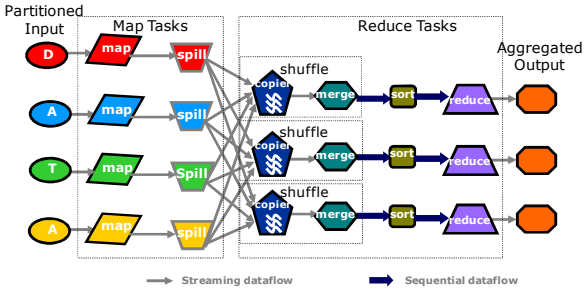


Figure 2. Dataflow graph of the Hadoop execution plan

Pig Script

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>1000000;
output = FOREACH big_groups GENERATE category,
          AVG(good_urls.pagerank);
```

Hive Query

```
SELECT category, AVG(pagerank)
FROM (SELECT category, pagerank, count(1) AS recordnum
          FROM urls WHERE pagerank > 0.2
          GROUP BY category) big_groups
WHERE big_groups.recordnum > 1000000
```

Figure 3. The Pig program [5] and Hive query example

In addition, the Pig and Hive systems allow the users to perform ad-hoc analysis of Big Data on top of Hadoop, using dataflow-style scripts and SQL-like queries respectively. For instance, Figure 3 shows the Pig program (an example in the original Pig paper [5]) and Hive query for the same operation (i.e., finding, for each sufficiently large category, the average pagerank of high-pagerank urls in that category). In these two systems, the logical dataflow graph of the operation is implicitly dictated by the language or query model, and

is automatically compiled into the physical execution plan (another dataflow graph) that is executed on the underlying Hadoop system.

Unlike the aforementioned systems that restrict their applications' dataflow graph, the Dryad system allows the users to specify an arbitrary directed acyclic graph to describe the application, as illustrated in Figure 4 (an example in the Dryad website [7]). The cloud runtime then refines the input dataflow graph and executes the optimized execution plan on the underlying cluster.

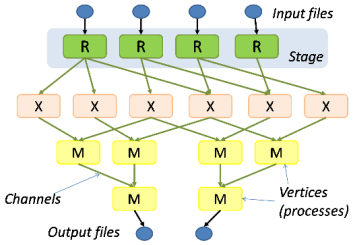


Figure 4. Dataflow graph of a Dryad application [7]

2.2 Motivations and Challenges

By exposing data parallelisms through the dataflow model and hiding the low level details of the underlying cluster, Big Data Cloud allows users to work at the appropriate level of abstraction, which makes it easy to develop and run highly scalable applications. Unfortunately, this abstraction makes it very difficult, if not impossible, for users to understand the cloud runtime behaviors. Consequently, it remains as a big challenge to efficiently provision and tune these massively distributed systems, which entails requesting and allocating the optimal number of (physical or virtual) resources, and optimizing the system and applications for better resource utilizations.

As Big Data Cloud grows in pervasiveness and scale, addressing this challenge becomes critically important (for instance, tuning Hadoop jobs is considered as a very difficult problem and requires a lot of efforts on understanding Hadoop internals in Hadoop community [8]; in addition, lack of tuning tools for Hadoop often forces users to resort to trial and error tuning [9]). This motivates our work to design tools that allows users to understand the runtime behaviors of Big Data applications, so that they can make educated decisions regarding how to improve the efficiency of these massively distributed systems – just as what traditional performance analyzers (e.g., gprof [10] and Intel VTune [11]) do for a single execution of a single program. Unfortunately, performance analysis for Big Data Cloud is particularly challenging due to its unique properties.

- *Massively distributed systems*: Each Big Data application is a complex distributed application, which may comprise tens of thousands of processes and threads running on thousands of machines. Understanding system behaviors in this context would require correlating concurrent performance activities (e.g., CPU cycles, retired instructions, lock contentions, etc.) across many programs and machines with each other.
- *High level abstractions*: Big Data Cloud allows users to work at an appropriately high level of abstraction, by hiding the messy details of parallelisms behind the dataflow model and dynamically instantiating the dataflow graph (including resource allocations, task scheduling, fault tolerance, etc.). Consequently, it is very difficult, if not impossible, for users to understand how the low level performance activities can be related to the high level abstraction (which they have used to develop and run their applications).

In this paper, we address these technical challenges through distributed instrumentations and dataflow-driven performance analysis. Our approach allows users to easily associate different low level performance activities with the high level dataflow model, and provide valuable insights into the runtime behaviors of Big Data Cloud and applications.

2.3 Goals and Non-Goals

Our goal is to design tools that help users to efficiently conduct performance analysis for Big Data Cloud. In particular, we want our tools to be broadly applicable to many different applications and systems, and to be applicable to even production systems (because it is often impossible to reproduce the cloud behaviors given the scale of Big Data Cloud). Several concrete design goals result from these requirements.

- *Low overhead*: It is critical that our tools have negligible (e.g., less than a few percent) performance impacts on the running applications.
- *No source code modifications*: Our tools should not require any modifications to the cloud runtime, middleware, messages, or applications.
- *Scalability*: Our tools need to handle applications that potentially comprise tens of thousands of processes/threads running on thousands of servers.
- *Extensibility*: We would like our tools to be flexible enough so that it can be easily extended to support different cloud systems.

We also have several non-goals.

- We are not developing tools that can replace the need for developers (e.g., by automatically

allocating the right amount of resources). Performance analysis for distributed systems is hard, and our goal is to make it easier for users, not to automate it.

- Our tools are not meant to verify the correct system behaviors, or diagnose the cause of faulty behaviors.

3. Overview of Our Approach

Our approach relies on distributed instrumentations on each node in the cloud, and then aggregating all the instrumentation results for dataflow-based analysis. The performance analysis framework consists of three major components, namely *tracker*, *aggregation engine* and *analysis engine*, as illustrated in Figure 5.

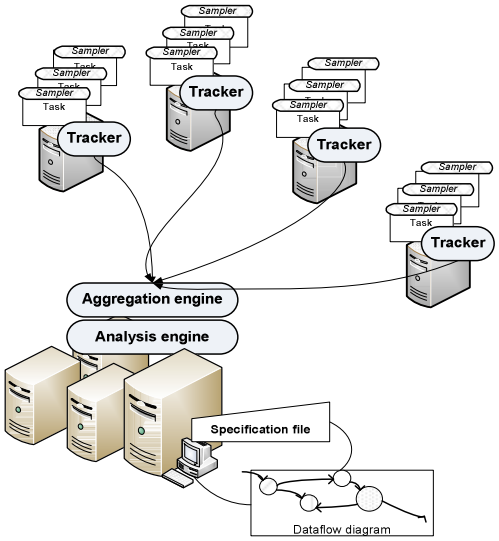


Figure 5. Performance analysis framework

Timestamp	Type	Target	Value

Figure 6. Format of the sampling record

The tracker is a lightweight agent running on every node. Each tracker has several *samplers*, which inspect the runtime information of the programs and system running on the local node (either periodically or based on specific events), and sends the sampling records to the aggregation engine. Each sampling record is of the format shown in figure 6.

- *Timestamp* is the sampling time for each record. Since the nodes in the cloud are usually in the same administrative domain and highly connected, it is easy to have all the nodes time-synchronized (e.g., in Hadoop all the slaves exchange *heartbeat* messages with the master periodically and the time synchronization information can be easily piggybacked); consequently the sampler can directly record its sampling time. Alternatively, the sampler can send the sampling record to the aggregation engine in real-time, which can then record the receiving time.
- *Type* specifies the type of the sampling record (e.g., CPU cycles, disk bandwidth, log files, etc.).
- *Target* specifies the source of the sampling record. It contains the name of the local node, as well as other sampler-specific information (e.g., CPUID, network interface name or log file name).
- *Value* contains the detailed sampling information of this record (e.g., CPU load, network bandwidth utilization, or a line/record in the log file).

The aggregation engine is responsible for collecting the sampling information from all the trackers in a distributed fashion, and storing the sampling information in a separate monitoring cluster for analysis. Any distributed log collection tools (e.g., Chukwa [12], Scribe [13] and Flume [14]) can be used as the aggregation engine. In addition, the analysis engine runs on the monitoring cluster, and is responsible for conducting the performance analysis and generating the analysis report, using the collected sampling information and a specification file describing the logical dataflow model of the specific Big Data cloud.

4. Dataflow-Based Performance Analysis

In order to help users to understand the runtime behaviors of Big Data Cloud, our framework presents the performance analysis results in the same dataflow model that is used in developing and running the applications. The key technical challenge is to re-construct the high level, dataflow-based, distributed and dynamic execution process for each Big Data application, based on the low level sampling records collected across different programs and machines. We address this challenge by:

1) Running a *task execution sampler* on every node to collect the execution information of each task in the application.
2) Describing the high level dataflow model of Big Data Cloud in a specification file provided to the analysis engine.
3) Constructing the dataflow execution process for the application based the dataflow specification and the program execution information.

4.1 Task Execution Sampler

To collect the program execution information, the task execution sampler instruments the cloud runtime and tasks running on the local node, and stores associated information into its sampling records at fixed time

intervals as follows.

- The *Target* field of the sampling record needs to store the identifier (e.g., application name, task name, process ID and/or thread ID) of the program that is instrumented to collect this piece of sampling information.
- The *Value* field of the sampling record must contain the *execution position* of the program (e.g., thread name, stack trace, basic-block ID and/or instruction address) at which the program is running when it is instrumented to collect this piece of sampling information.

In practice, the task execution sampler can be implemented using any traditional instrumentation tool (which runs on a single machine), such as Intel VTune.

4.2 Dataflow Specification

In order for the analysis engine to efficiently conduct the dataflow-based analysis, the dataflow specification needs to describe not only the dataflow graph (e.g., vertices and edges), but also the high level resource mappings of the dataflow model (e.g., physical implementations of vertices/edges, parallelisms between different phases/stages, and communication patterns between different stages). Consequently, the dataflow specification does require *a priori* knowledge of the cloud system. On the other hand, the users are not required to write the specification; instead, the dataflow specification is provided by the cloud system or the performance analyzer, and is either written by the developers of the cloud system (and/or the performance analyzer), or dynamically generated by the cloud runtime (e.g., by the Hive query compiler). The format of the dataflow specification is illustrated in Figure 7 and described in detail below.

- The *Input* (*Output*) section contains a list of <*inputId: storage location*> (<*outputId: storage location*>), where the storage location specifics which storage system (e.g., HDFS or MySQL) is used to store the input (output) data.

- The *Vertices* section contains a list of <*vertexId: program location*>, where the *program location* specifies the portion of program (e.g., the corresponding thread or function) running on this graph vertex (i.e., processing stage). It is required that each *execution position* collected by the task execution sampler can be mapped to a unique program location in the specification, so that the analysis engine can determine which vertex each task execution sampling record belongs to.

```
//dataflow graph
Input { //list of <inputId: storage location>
   In1: storage location
   …
}
Output { //list of <outputId: storage location>
   Out1: storage location
   …
}
Vertices { //list of <vertexId: program location>
   V1: program location
   …
}
Edges { //list of <edgeId: inputId/vertexId→vertexId/outputId>
   E1: In1→V1
   E2: V1→V2
   …
}
//resource mapping
Vertex Mapping { //list of Task Pool
   Task Pool [(name)] <(cardinality)> { //ordered list of Phase
      Phase [(name)] { //list of Thread Pool or Thread Group Pool
         Thread Pool [(name)] <(cardinality)> {
            //ordered list of vertexId
            V1, V2, …
         } //end of Thread Pool
         Thread Group Pool [(name)] <(cardinality, group size)> {
            //a single vertexId
            V3
         } //end of Thread Group Pool
         ..
      } //end of Phase
      Phase [(name)] { … }
      …
   } //end of Task Pool
   Task Pool [(name)] <(cardinality)> { … }
   …
}
Edge Mapping { //list of <edgeId: edge type, endpoint location>
   E1: edge type, endpoint location
   …
}
```

Figure 7. Dataflow specification of Big Data Cloud

- The *Edges* section contains a list of <*edgeId: inputId/vertexId→vertexId/outputId*>, which defines all the graph edges (i.e., communication channels).

- The *Vertex Mapping* section describes the high level resource mappings and parallelisms of the graph vertices. This section contains a list of *Task Pool* subsections; for each *Task Pool* subsection, the cloud runtime will launch several tasks (or processes) that can potentially run on the different nodes in parallel. The *Task Pool* subsection contains an ordered list of *Phase* subsections, and each task belonging to this task pool will sequentially execute these phases in the specified order.

- The *Phase* subsection contains a list of *Thread Pool* or *Thread Group Pool* subsections; for each of these subsections, the associated task will spawn several

threads or thread groups in parallel. The *Thread Pool* subsection contains an ordered list of *vertexId*, and each thread belonging to this thread pool will sequentially execute these vertices in the specified order. On the other hand, a number of threads (as determined by *group size*) in the thread group will run in concert with each other, executing the vertex specified in the *Thread Group Pool* subsection.

- The *cardinality* of the *Task Pool*, *Thread Pool* or *Thread Group Pool* subsections determines the numbers of instances (i.e., processes, threads or thread groups) to be launched. It can have several values as follows.
  (1) *N* – exactly *N* instances will be launched.
  (2) *1~N* – up to *N* instances will be launched.
  (3) *1~∞* – the number of instances to be launched is dynamically determined by the cloud runtime.

- The *Edge Mapping* section contains a list of <*edgeId: edge type, endpoint location*>. The edge type specifies the physical implementation of the edge, such as network connection, local file or memory buffer. The endpoint location specifies the communication patterns between the vertices, which can be *intra-thread/intra-task/intra-node* (i.e., data transfer exists only between vertex instances running in the same thread, the same task and the same node respectively), or *unconstrained*.

It is possible to extend the specification to support even more complex dataflow model and resource mappings (e.g., *Process Group Pool*); however, the current model is sufficient for all the Big Data cloud infrastructures that we have considered. For instance, the dataflow specification for our Hadoop cluster is shown in Figure 8.

4.3 Dataflow-Based Analysis

As described in the previous sections, the program execution information collected by task execution samplers is generic in nature, and the dataflow model of the specific Big Data cloud is defined in a specification file. Based on these data, the analysis engine can re-construct the dataflow execution process for the Big Data applications, and associate different performance activities with the high level dataflow model. In this way, our framework can be easily applied to different cloud systems by simply changing the specification file. We defer the detailed description of the dataflow construction algorithm to section 5.3.

```
//Hadoop dataflow graph
Input { //list of <inputId: storage location>
   Input:HDFS
}
Output { //list of <outputId: storage location>
   Output:HDFS
}
Vertices { //list of <vertexId: program location>
   map:     MapTask.run
   spill:   SpillThread.run
   copier:  MapOutputCopier.run
   merge:   InMemFSMergeThread.run or
            LocalFSMerger.run
   sort:    ReduceCopier.createKVIterator#ReduceCopier.access
   reduce:  runNewReducer or runOldReducer
}
Edges { //list of <edgeId: inputId/vertexId →vertexId/outputId>
   E1: Input→map
   E2: map→spill
   E3: spill→copier
   E4: copier→merge
   E5: merge→sort
   E6: sort→reduce
   E7: reduce→Output
}
Vertex Mapping { //list of Task Pool
   Task pool (Map) (1~∞) { //ordered list of Phase
      Phase { //list of Thread Pool or Thread Group Pool
         Thread Pool (1) {map}
         Thread Pool (1) {spill}
      }
   }
   Task Pool (Reduce) (1~∞) {
      Phase (shuffle) {
         Thread Group Pool (copy) (1, 20) {copier}
         Thread Group Pool (merge) (1, 2) {merge}
      }
      Phase { Thread Pool (1) {sort, reduce} }
   }
}
Edge Mapping { //list of <edgeId: edge type, endpoint location>
   E1: HDFS, unconstrained
   E2: memory buffer, intra-task
   E3: HTTP, unconstrained
   E4: memory buffer, intra-task
   E5: memory buffer or local file, intra-task
   E6: memory buffer or local file, intra-thread
   E7: HDFS, unconstrained
}
```

Figure 8. Hadoop dataflow specification

5. HiTune: A Dataflow-Based Hadoop Performance Analyzer

Based on our general performance analysis framework, we have implemented HiTune, a scalable, lightweight and extensible performance analyzer for Hadoop. In this section, we describe the implementation of HiTune, and in particular, how it is carefully engineered to meet our design goals that are described in section 2.3.

5.1 Implementation of Tracker

In our current implementation, all the nodes in the

Hadoop cluster are time synchronized (e.g., using a time service), and a tracker runs on each node in the cluster. Currently, the tracker consists of three samplers (i.e., *task execution sampler*, *system sampler* and *log file sampler*), and each sampler directly stores the sampling time in the *Timestamp* field of its sampling record.

We have chosen to implement the task execution sampler using binary instrumentation techniques, so that it can instrument and collect the program execution information without any source code modifications. Specifically, the task execution sampler runs as a Java programming language agent [15]; whenever the Hadoop framework launches a JVM to be instrumented, it dynamically attaches to the JVM the sampler agent, which samples the Java thread stack trace and state for all the threads in the JVM at specified intervals (during the entire lifetime of the JVM).

For each sampling record generated by the task execution sampler, its *Target* field contains the node name, the task name and the Java thread ID; and its *value* field contains the current execution position (i.e., the Java thread name and thread stack trace) as well the as the Java thread state. That is, the *Target* field is specified using the identifier of the runtime program instance (i.e., the thread ID), which allows the analysis engine to construct the entire sampling trace of each thread; in addition, the execution position is specified using the static program names (i.e., the Java thread name and method names), which allows the dataflow model and resource mappings to be easily described in the specification file.

In addition, the system sampler simply reports the system statistics (e.g., CPU load, disk I/O, network bandwidth, etc.) periodically using the *sysstat* package, and the log sampler reports the Hadoop log information (including Hadoop metrics and job history files) whenever there is new log information.

Since the tracker (especially the task execution sampler) needs to instrument the Hadoop tasks running on each node, it is the major source of performance overheads in HiTune. We have carefully designed and implemented the tracker (e.g., the task execution sampler caches the stack traces in memory and batches the write-out to the aggregation engine), so that it has very low (less than 2% according to our measurement) performance impacts on Hadoop applications.

5.2 Implementation of Aggregation Engine

To ensure its scalability, the aggregation engine is implemented as a distributed data collection system, which can collect the sampling information from potentially thousands of nodes in the Hadoop cluster. In the current implementation, we have chosen to use Chukwa (a distributed log collection framework) as our aggregation engine. Every sampler in HiTune directly sends its sampling records to the *Chukwa agent* running on the local node, which in turn sends data to the *Chukwa collectors* over the network; the collector is responsible for storing the sampling data in a (separate) monitoring Hadoop/HDFS cluster.

5.3 Implementation of Analysis Engine

The sampling data for a Hadoop job can be potentially very large in size (e.g., about 100GB for TeraSort [16][17] in our cluster). We address the scalability challenge by first storing the sampling data in HDFS (as described in section 5.2), and then running the analysis engine as a Hadoop application on the monitoring Hadoop cluster in an offline fashion.

Based on the *Target* field (i.e., the node name, the task name and the Java thread ID) of every task execution sampling record, the analysis engine first constructs a sampling trace for each thread (i.e., the sequence of all task execution sampling records belonging to that thread, ordered by the record timestamps) in the Hadoop job.

The program location (used in the dataflow specification) can therefore be defined as a range of consecutive sampling records in one thread trace (or, in the case of thread group, multiple ranges each in a different thread). Each record range is identified by the starting and ending records, which are specified using their execution positions (i.e., partial stack traces). For instance, all the records between the first appearance and the last appearance of *MapTask.run* (or simply the *MapTask.run* method) constitute one instance of the *map* vertex. See Figure 8 for the detailed dataflow specification of our Hadoop cluster.

Based on the *Target* and *Timestamp* fields of the two boundary records of corresponding program locations, the analysis engine then determines which machine each instance of a vertex runs on, when it starts and when it ends. Finally, it associates all the system and log file sampling records to each instance of the dataflow graph vertex (i.e., the processing stage), again using the *Target* and *Timestamp* information of the records.

With the algorithm and dataflow specification described above, the analysis engine can easily reconstruct the

dataflow execution for the Hadoop job and associates different sampling records with the dataflow graph. In addition, the performance analysis algorithm is itself implemented as a Hadoop application, which processes the sampling records for each JVM simultaneously.

Consequently, we can generate various analysis reports that provide valuable insights into the Hadoop runtime behaviors (presented in the same dataflow model used in developing and running the Hadoop job). For instance, a timeline based execution chart for all task instances, similar to the pipeline space-time diagram [18], can be generated so that users can get a complete picture about the dataflow-based execution process of the Hadoop job. It is also possible to generate the hotspot breakdown (e.g., disk I/O vs. network transfer vs. computations) for each vertex in the dataflow, so that users can identify the possible bottlenecks in the Hadoop cluster. We show some analysis reports and how they are used to help our Hadoop performance analysis and tuning in section 7.

6. Evaluations

In this section, we experimentally evaluate the runtime overheads and scalability of HiTune, using three benchmarks (namely, Sort, WordCount and Nutch indexing) in the *HiBench* Hadoop benchmark suite [17], as shown in Table 1. The Hadoop cluster used in our experiment consists of one master (running JobTracker and NameNode) and up to 20 slaves (running TaskTracker and DataNode); the detailed server configurations are shown in Table 2. Every server has two GbE NICs, each of which is connected to a different gigabit Ethernet switch, forming two different networks; one network is used for the Hadoop jobs, and the other is used for administration and monitoring tasks (e.g., the Chukwa aggregation engine in HiTune).

Table 1. Hadoop benchmarks

Benchmark	Input Data
Sort	60GB generated by *RandomWriter* example.
WordCount	60GB generated by *RandomTextWriter* example
Nutch indexing	19GB data generated by crawling an in-house Wikipedia mirror

Table 2. Server configurations

Processor	Dual-socket quad-core Intel Xeon processor
Disk	4 SATA 7200RPM HDDs
Memory	24 GB ECC DRAM
Network	2 Gigabit Ethernet NICs
OS	Redhat Enterprise Linux 5.4

We evaluate the runtime overheads of HiTune by measuring the Hadoop performance (speed and throughput) as well as the system resource (e.g., CPU and memory) utilizations of the Hadoop cluster. The speed is measured using the job running time, and the throughput is defined as the number of tasks completed per minute when the Hadoop cluster is at full utilization (by continuously submitting multiple jobs to the cluster). In addition, we evaluate the scalability of HiTune by analyzing that, when there are more nodes in the Hadoop cluster, whether the runtime overheads increase and whether it becomes more complex for HiTune to conduct the dataflow-based performance analysis.

6.1 Runtime Overheads

As mentioned in section 5.1, the tracker (especially the task execution sampler) is the major source of runtime overheads in HiTune. This is because the task execution sampler needs to instrument the Hadoop tasks running on each node, while the aggregation and analysis of sampling data are performed on a separate monitoring cluster in an offline fashion.

We first compare the instrumented Hadoop performance (measured when the tracker is running) and the baseline performance (measured when the tracker is completely turned off). In our experiment, when the tracker is running, the task execution sampler dumps the Java thread stack trace every 20 milliseconds, the system sampler reports the system statistics every 5 seconds, and the Hadoop cluster is configured to output its metrics to the log file every 10 seconds. Figures 9 and 10 show the ratio of the instrumented performance over the baseline performance for job running time (lower is better) and throughput (higher is better) respectively. It is clear that the overhead of running the tracker is very low in terms of performance – the instrumented job running time is longer than the baseline by less than 2%, and the instrumented throughput is lower than the baseline by less than 2%.

In addition, we also compare the *instrumented system resource utilizations* (measured when the tracker is running) and the *baseline utilizations* (measured when only the system sampler is running, which is needed to report the system resource utilizations periodically). Since the sampling records are aggregated using a separate network, we only present the CPU and memory utilization results of the Hadoop cluster in this paper. Figures 11 and 12 show the ratio of the instrumented CPU and memory utilizations over the baseline utilizations respectively. It is clear that the overhead of running the tracker is also very low in terms of resource utilizations – either the instrumented CPU or memory utilization is higher than the baseline by less than 2%.

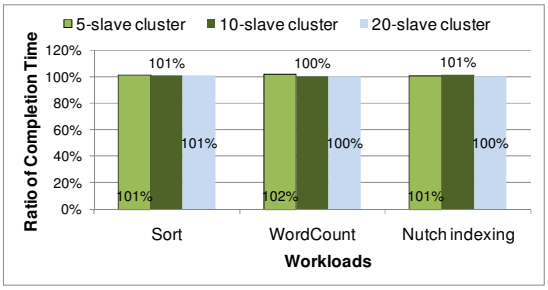


Figure 9. Ratio of instrumented job running time over baseline job running time

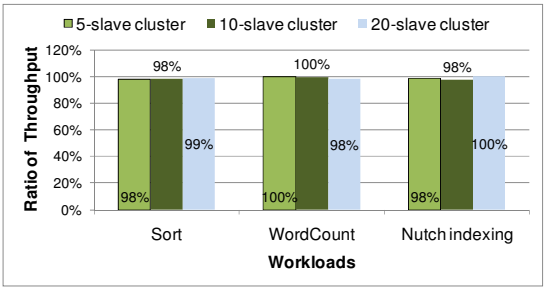


Figure 10. Ratio of instrumented cluster throughput over baseline cluster throughput

Figure 11. Ratio of instrumented cluster CPU utilization over baseline cluster CPU utilization

Figure 12. Ratio of instrumented cluster memory utilization over baseline cluster memory utilization

In summary, HiTune is a very lightweight performance analyzer for Hadoop, with very low (less than 2%) runtime overheads in terms of speed, throughput and system resource utilizations. In addition, HiTune scales very well in terms of the runtime overheads, because it instruments each node in the cluster independently and consequently the runtime overheads remain the same even when there are more nodes in the cluster (as confirmed by the experimental results).

6.2 Complexity of Performance Analysis

Since the analysis engine needs to re-construct the dataflow execution of a Hadoop job and associate the sampling records to each vertex instance in the dataflow, the complexity of analysis can be evaluated by comparing the sizes of sampling data and the numbers of vertex instances between different sized clusters.

Figure 13 shows the sampling data sizes for the 5-, 10- and 20-slave clusters. It is clear that the sampling data sizes remain about the same (or increase very slowly) for different sized clusters (e.g., only less than 18% increase in the sample data size even when the cluster size is increased by 4x). Intuitively, since HiTune samples each instance of the processing stages at fixed time intervals, the sampling data size is proportional to the sum of the running time of all vertex instances. As long as the underlying Hadoop framework scales well with the cluster sizes, the sum of the vertex instance running time will remain about the same (or increase very slowly), and so does the sampling data size. In practice, even with very large (1000s of nodes) clusters, a MapReduce job usually runs on about 100s of worker machines [19], and the Hadoop framework scales reasonably well with that number (100s) of machines.

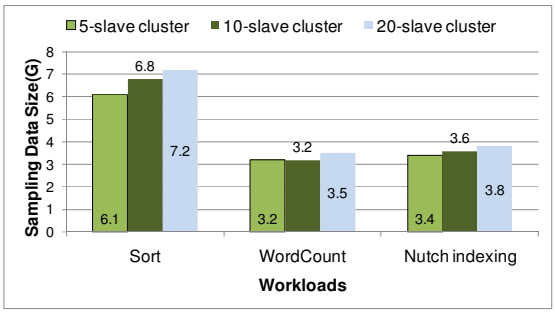


Figure 13. Comparison of sampling data sizes

In addition, assume *M* and *R* are the total numbers of the map and reduce tasks of a Hadoop job respectively. Since in the Hadoop dataflow model (as shown in Figure 8) each map task contains two stages and each reduce task contains four stages, the total number of vertex instances can be computed as *2*M+4*R*. In practice, the number of map tasks is about 26x of that of reduce tasks in average for each MapReduce job [20], and therefore the vertex instance count is about *2.15*M*. Since the number of map tasks (*M*) of a Hadoop job is typically determined by its input data size (e.g., by the number of HDFS file blocks), the number of vertex instances will also remain about the same for different sized clusters in practice.

In summary, the complexity for HiTune to conduct the dataflow-based performance analysis will remain about the same even when there are more nodes in the cluster (or, more precisely, the dataflow-based performance analysis in HiTune scales as well as Hadoop does with the cluster sizes), because the sampling data sizes and the vertex instance counts will remain about the same even when there are more nodes in the cluster. In addition, we have implemented the analysis engine as a Hadoop application, so that the dataflow-based performance analysis can be parallelized using another monitoring Hadoop cluster. For instance, to process the 100GB sampling data generated when running TeraSort in our cluster, it takes about 16 minutes on a single-slave monitoring cluster, and about 5 minutes on a 4-slave monitoring cluster.

7. Experience

HiTune has been used intensively inside Intel for Hadoop performance analysis and tuning (e.g., see [17]). In this section, we share our experience on how we use HiTune to efficiently conduct performance analysis and tuning for Hadoop, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches (e.g., system statistics, Hadoop logs and metrics, and traditional profiling).

7.1 Tuning Hadoop Framework

One performance issue we encountered is extremely low system utilizations when sorting many small files (3200 500KB-sized files) using Hadoop 0.20.1 – system statistics collected by the cluster monitoring tools (e.g., Ganglia [21]) show that the CPU, disk I/O and network bandwidth utilizations are all below 5%. That is, there are no obvious bottlenecks or hotspots in our cluster; consequently, traditional tools (e.g., system monitors and program profilers) fail to reveal the root cause.

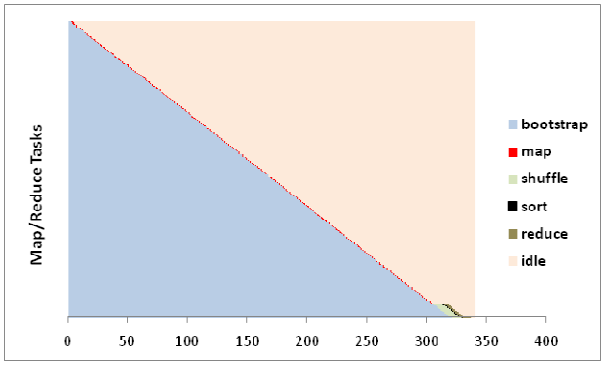


Figure 14. Dataflow execution for sorting many small files with Hadoop 0.20.1

To address this performance issue, we used HiTune to reconstruct the dataflow execution process of this Hadoop job, as illustrated in Figure 14. The x-axis represents the elapse of wall clock time, and each horizontal line in the chart represents a map or reduce task. Within each line, *bootstrap* represents the period before the task is launched, *idle* represents the period after the task is complete, *map* represents the period when the map task is running, and *shuffle*, *sort* and *reduce* represent the periods when (the instances of) the corresponding stages are running respectively.

As is obvious in the dataflow execution, there are few parallelisms between the Map tasks, or between the Map tasks and Reduce tasks in this job. Clearly, the task scheduler in Hadoop 0.20.1 (*Fair Scheduler* [22] is used in our cluster) fails to launch all the tasks as soon as possible in this case. Once the problem is isolated, we quickly identified the root cause – by default, the Fair Scheduler in Hadoop 0.20.1 only assigns one task to a slave at each heartbeat (i.e., the periodical keep-alive message between the master and slaves), and it schedules map tasks first whenever possible; in our job, each map task processes a small file and completes very fast (faster than the heartbeat interval), and consequently each slave runs the map tasks sequentially and the reduce tasks are scheduled after all the map tasks are done.

To fix this performance issue, we upgraded the cluster to *Fair Scheduler 2.0* [23][24], which by default schedules multiple tasks (including reduce tasks) in each heartbeat; consequently the job runs about 6x faster (as shown in Figure 15) and the cluster utilization is greatly improved.

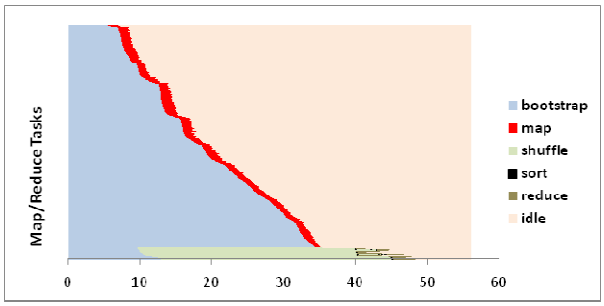


Figure 15. Dataflow execution for sorting many small files with Fair Scheduler 2.0

7.2 Analyzing Application Hotspots

In the previous section, we demonstrate that the high level dataflow execution process of a Hadoop job helps users to understand the dynamic task scheduling and assignment of the Hadoop framework. In this section,

we show that the dataflow execution process helps users to identify the data shuffle gaps between map and reduce, and that relating the low level performance activities to the high level dataflow model allows users to conduct fine-grained, dataflow-based hotspot breakdown (so as to understand the hotspots of the massively distributed applications).

Figure 16 shows the dataflow execution, as well as the timeline based CPU, disk and network bandwidth utilizations of TeraSort [16][17] (sorting 10 billion 100-byte records). It has high CPU utilizations during the map tasks, because the map output data are compressed (using the default codec in Hadoop) to reduce the disk and network I/O. (Compressing the input or output of TeraSort is not allowed in the benchmark specs).

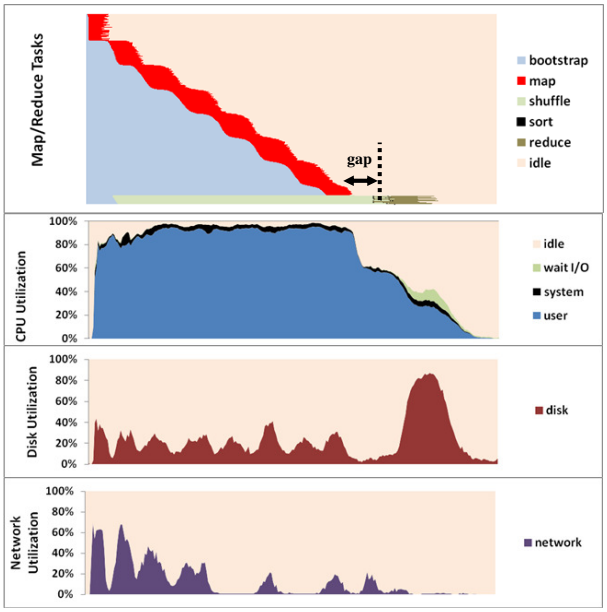


Figure 16. TeraSort (using default compression codec)

However, the dataflow execution process of TeraSort shows that there is a large gap (about 15% of the total job running time) between the end of map tasks and the end of shuffle phases. According to the communication patterns specified in the Hadoop dataflow model (see Figure 2 and Figure 8), shuffle phases need to fetch the output from all the map tasks in the copier stages, and ideally should complete as soon as all the map tasks complete. Unfortunately, traditional tools or Hadoop logs fail to reveal the root cause of the large gap, because during that period, none of the CPU, disk I/O and network bandwidth are bottlenecked, the "*Shuffle Fetchers Busy Percent*" metric reported by the Hadoop framework is always 100%, while increasing the number of *copier* threads does not improve the utilization or performance.

To address this issue, we used HiTune to conduct hotspot breakdown of the shuffle phases, which is possible because HiTune has associated all the low level sampling records with the high level dataflow execution of the Hadoop job. The dataflow-based hotspot breakdown (see Figure 17) shows that, in the shuffle stages, the *copier* threads are actually idle 80% of the time, waiting (in the *ShuffleRamManager.reserve* method) for the occupied memory buffers to be freed by the *memory merge* threads. (The idle vs. busy breakdown and the method hotspot are determined using the Java thread state and stack trace in the task execution sampling records respectively). On the other hand, most of the busy time of the *memory merge* thread is due to the compression, which is the root cause of the large gap between map and shuffle. To fix this issue and reduce the compression hotspots, we changed the compression codec to *LZO* [25], which improves the TeraSort performance by more than 2x and completely eliminates the gap (see Figure 18).

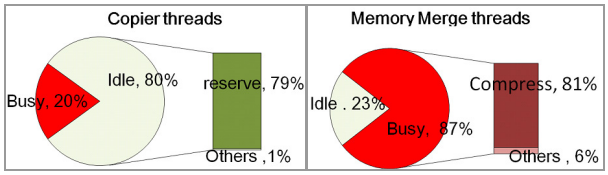


Figure 17. Copier and Memory Merge threads breakdown (using default compression codec)

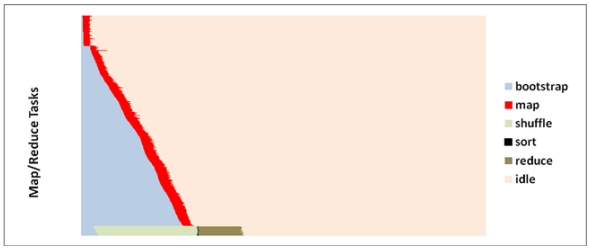


Figure 18. TeraSort (using *LZO* compression)

7.3 Diagnosing Hardware Problems

By examining Figure 18 in more detail, we also found that the reduce stage running time is significantly skewed among different reduce tasks – a small number of stages are much slower than the others, as shown in Figure 19.

Figure 19. Reduce tasks of TeraSort (using *LZO* compression)

Based on the association of the low level sampling records and the high level dataflow model, we use HiTune to generate the normalized average running time and the idle vs. busy breakdown of the reduce stages (grouped by the Tasktrackers that the stages run on) in Figure 20. It is clear that reduce stages running on the 3rd and 7th TaskTrackers are much slower (about 20% and 14% slower than the average respectively). In addition, while all the reduce stages have about the same busy time, the reduce stages running on these two TaskTrackers have more idle time, waiting in the *DFSOutputStream.writeChunk* method (i.e., writing data to HDFS). Since the data replication factor in TeraSort is set to 1 (as required by the benchmark specs), the HDFS write operations in the reduce stage only writes to the local disks. By examining the average write bandwidth of the disks on these two TaskTrackers, we finally identified the root cause of this problem – there is one disk on each of these two nodes that is much slower than other disks in the cluster (about 44% and 30% slower than the average respectively), which is later confirmed to have bad sectors through a very expensive *fsck* process.

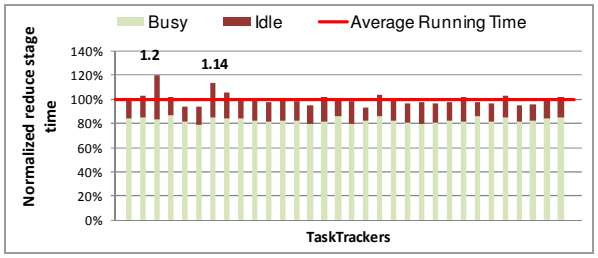


Figure 20. Normalized average running time and busy vs. idle breakdown of reduce stages

7.4 Extending HiTune to Other Systems

Since the initial release of HiTune inside Intel, it has been extended by the users in different ways to meet their requirements. For instance, new samplers are added so that processor microarchitecture events and power state behaviors of Hadoop jobs can be analyzed using the dataflow model.

In addition, HiTune has also been applied to Hive (an open source data warehouse built on top of Hadoop), by extending the original Hadoop dataflow model to include additional phases and stages, as illustrated in Figure 21. The map stage is divided into 5 smaller stages – namely, *Stage Init*, *Hive Init*, *Hive Active*, *Hive Close* and *Stage close*; in addition, the reduce stage is divided into 4 smaller stages – namely, *Hive Iinit*, *Hive Active*, *Hive Close* and *Stage Close*. This is accomplished by providing to the analysis engine a new specification file that describes the dataflow model and resource mappings in Hive.

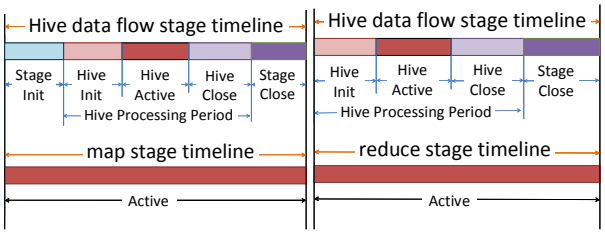


Figure 21. Extended dataflow model for Hive

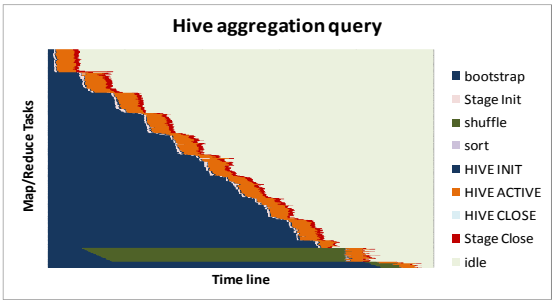


Figure 22. Dataflow execution of the Hive query

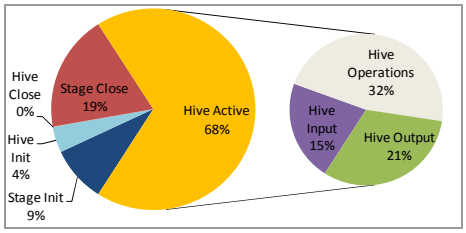


Figure 23. Map stage breakdown

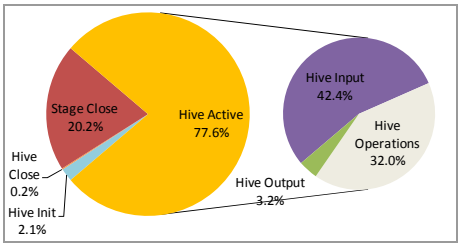


Figure 24. Reduce stage breakdown

Figure 22 shows the dataflow execution process for the aggregation query in Hive performance benchmarks [26][9]. In addition, Figures 23 and 24 show the dataflow-based breakdown of the map/reduce stages for the aggregation query (both the map and reduce Hive active stages are further broken into 3 portions: *Hive Input*, *Hive Operation* and *Hive Output* based on the Java methods). As shown in Figures 23 and 24, the query spends only about 32% of its time performing the Hive Operations; on the other hand, it spends about 68% of its time on the data input/output, as well as the initialization and cleanup of the Hadoop/Hive frameworks. Therefore, to optimize this Hive query, it is more critical to reduce the size of intermediate results,

to improve the efficiency of data input/output, and to reduce the overheads of the Hadoop/Hive frameworks.

8. Related Work

There are several distributed system tracing tools (e.g., Magpie [27], X-Trace [28] and Dapper [29]), which associates and propagates the tracing metadata as the request passes through the system. With this type of path information, the tracing tools can easily construct an event graph capturing the causality of events across the system, which can be then queried for various analyses [30]. Unfortunately, these tools would require changes not only to source codes but also to message schemas, and are usually restricted to a small portion of the system in practice (e.g., Dapper only instruments the threading, callback and RPC libraries in Google [29]). In contrast, our approach uses binary instrumentations to sample the tasks in a distributed and independent fashion at each node, and reconstructs the dataflow execution process of the application based on *a priori* knowledge of Big Data Cloud. Consequently, it requires no modifications to the system, and therefore can be applied more extensively to obtain richer information (e.g., the hottest function) than these tracing tools.

Our distributed instrumentations are similar to Google-Wide Profiling (GWP) [31], which samples across machines in multiple data centers for production applications. In addition, the current Hadoop framework can profile specific map/reduce tasks using traditional Java profilers (e.g., HPROF [32]), which however have very high overheads and are usually applied to a small (2 or 3) number of tasks. More importantly, both GWP and the existing profiling support in Hadoop focus on providing traditional performance analysis to the distributed systems (e.g., by allowing the users to directly query the low level sampling data). In contrast, our key technical challenge is to reconstruct the high level dataflow execution of the application based on the low level sampling data, so that users can work on the same dataflow model used in developing and running their Big Data applications.

In the industry, traditional cluster monitoring tools (e.g., Ganglia [21], Nagios [33] and Cacti [34]) have been widely used to collect system statistics (e.g., CPU load) from all the machines in the cluster; in addition, several large-scale log collection systems (e.g., Chukwa [12], Scribe [13] and Flume [14]) have been recently developed to aggregate log data from a large number of servers. All of these tools focus on providing a distributed framework to collect statistics and logs, and are orthogonal to our work (e.g., we have actually used Chukwa as the aggregation engine in the current HiTune implementation).

Existing diagnostic tools for Hadoop and Dryad (e.g., Vaidya [35], Kahuna [36] and Artemis [37]) focus on mining the system logs to detect performance problems. For instance, it is possible to construct the task execution chart (as shown in section 7.1) using the Hadoop job history files. Compared to these tools, our approach (based on distributed instrumentation and dataflow-driven performance analysis) has many advantages. First, it can provide much more insights, such as dataflow-based hotspot breakdown (see sections 7.2 and 7.3), into the cloud runtime behaviors. More importantly, performance problems of massively distributed systems are very complex, and are often due to issues that the developers are completely unaware of (and therefore are not exposed by the existing codes or logs). For instance, in section 7.2, the Hadoop framework shows that the shuffle fetchers are always busy, while detailed breakdown provided by HiTune reveals that copiers are actually idle most of the time. Finally, our approach is much more general, and consequently can be easily extended to support other systems such as Hive (see section 7.4).

9. Conclusions

In this paper, we propose a general approach of performance analysis for Big Data Cloud, based on distributed instrumentations and dataflow-driven performance analysis. Based on this approach, we have implemented HiTune (a Hadoop performance analyzer), which provide valuable insights into the Hadoop runtime behaviors with every low overhead, no source code changes, very good scalability and extensibility. We also report our experience on how to use HiTune to efficiently conduct performance analysis and tuning for Hadoop, demonstrating the benefits of dataflow-based analysis and the limitations of existing approaches.

Reference

[1] D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, H. Liu. “Data warehousing and analytics infrastructure at facebook”. The 36th ACM SIGMOD International Conference on Management of Data, 2010.

[2] J. Dean, S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. The 6th Symposium on Operating Systems Design and Implementation, 2004.

[3] Hadoop. http://hadoop.apache.org/

[4] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly. “Dryad: Distributed Data-Parallel Programs from

Sequential Building Blocks". The 2nd European Conference on Computer Systems, 2007.
[5] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins. "Pig latin: a not-so-foreign language for data processing". The 34th ACM SIGMOD international conference on Management of data, 2008.
[6] A. Thusoo, R. Murthy, J. S. Sarma, Z. Shao, N. Jain, P. Chakka, S. Anthony, H. Liu, N. Zhang. "Hive - A Petabyte Scale Data Warehousing Using Hadoop". The 26th IEEE International Conference on Data Engineering, 2010.
[7] Dryad. http://research.microsoft.com/en-us/projects/Dryad/
[8] "Hadoop and HBase at RIPE NCC". http://www.cloudera.com/blog/2010/11/hadoop-and-hbase-at-ripe-ncc/
[9] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, M. Stonebraker. "A comparison of approaches to large-scale data analysis". The 35th SIGMOD international conference on Management of data, 2009.
[10] S. L. Graham, P. B. Kessler, M. K. Mckusick. "Gprof: A call graph execution profiler". The 1982 ACM SIGPLAN Symposium on Compiler Construction, 1982.
[11] Intel VTune Performance Analyzer. http://software.intel.com/en-us/intel-vtune/
[12] A. Rabkin, R. H. Katz. "Chukwa: A system for reliable large-scale log collection", Technical Report UCB/EECS-2010-25, UC Berkeley, 2010.
[13] Scribe. http://github.com/facebook/scribe
[14] Flume. https://github.com/cloudera/flume
[15] Java Instrumentation. http://download.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html
[16] Sort benchmark. http://sortbenchmark.org/
[17] S. Huang, J. Huang, J. Dai, T. Xie, B. Huang. "The HiBench Benchmark suite: Characterization of the MapReduce-Based Data Analysis". IEEE 26th International Conference on Data Engineering Workshops, 2010.
[18] J. L. Hennessy, D. A. Patterson. "Computer Architecture: A Quantitative Approach". Morgan Kaufmann, 4th edition, 2006.
[19] Jeff Dean. "Designs, Lessons and Advice from Building Large Distributed Systems". The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.
[20] Jeffrey Dean. "Experiences with MapReduce, an abstraction for large-scale computation". The 15th International Conference on Parallel Architectures and Compilation Techniques, 2006.
[21] Ganglia. http://ganglia.sourceforge.net/
[22] A fair sharing job scheduler. https://issues.apache.org/jira/browse/HADOOP-3746
[23] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, I. Stoica. "Job Scheduling for Multi-User MapReduce Clusters". Technical Report UCB/EECS-2009-55, UC Berkeley, 2009.
[24] Hadoop Fair Scheduler Design Document. https://svn.apache.org/repos/asf/hadoop/mapreduce/trunk/src/contrib/fairscheduler/designdoc/fair_scheduler_design_doc.pdf
[25] Hadoop LZO patch. http://github.com/kevinweil/hadoop-lzo
[26] Hive performance benchmark. https://issues.apache.org/jira/browse/HIVE-396
[27] P. Barham, R. Isaacs, R. Mortier, D. Narayanan. "Magpie: online modelling and performance-aware systems". The 9th conference on Hot Topics in Operating Systems, 2003.
[28] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, I. Stoica. "X-trace: A pervasive network tracing framework". The 4th USENIX Symposium on Networked Systems Design & Implementation, 2007.
[29] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag. "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure". Google Research, 2010.
[30] B. Chun, K. Chen, G. Lee, R. Katz, S. Shenker. "D3: Declarative Distributed Debugging". Technical Report UCB/EECS-2008-27, UC Berkeley, 208.
[31] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, R. Hundt. "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers". IEEE Micro (2010), pp. 65-79.
[32] "HPROF: a Heap/CPU Profiling Tool in J2SE 5.0". http://java.sun.com/developer/technicalArticles/Programming/HPROF.html
[33] Nagios. http://www.nagios.org/
[34] Cacti. http://www.cacti.net/
[35] V. Bhat, S. Gogate, M. Bhandarkar. "Hadoop Vaidya". Hadoop World 2009.
[36] J. Tan, X. Pan, S. Kavulya, R. Gandhi, P. Narasimhan. "Kahuna: Problem Diagnosis for MapReduce-Based Cloud Computing Environments". IEEE/IFIP Network Operations and Management Symposium (NOMS), 2010.
[37] G. Cretu, M. Budiu, M. Goldszmidt . "Hunting for problems with Artemis". First USENIX conference on Analysis of system logs, 2008.

Taming the Flying Cable Monster: A Topology Design and Optimization Framework for Data-Center Networks

Jayaram Mudigonda* Praveen Yalagandula* Jeffrey C. Mogul*
Jayaram.Mudigonda@hp.com Praveen.Yalagandula@hp.com Jeff.Mogul@hp.com
*HP Labs, Palo Alto, CA 94304

Abstract

Data-center network designers now have many choices for high-bandwidth, multi-path network topologies. Some of these topologies also allow the designer considerable freedom to set parameters (for example, the number of ports on a switch, link bandwidths, or switch-to-switch wiring patterns) at design time. This freedom of choice, however, requires the designer to balance among bandwidth, latency, reliability, parts cost, and other real-world details.

Designers need help in exploring this design space, and especially in finding optimal network designs. We describe the specific challenges that designers face, and we present *Perseus*, a framework for quickly guiding a network designer to a small set of candidate designs. We identify several optimization problems that can be accommodated within this framework, and present solution algorithms for many of these problems.

1 Introduction

Imagine that you have been given the task to design a shipping-container ("pod") cluster containing 1000 server-class computers. The container provides the necessary power and cooling, you have already chosen the servers, and now you must choose a network to connect the servers within the pod.

Suppose that you know the pod will be used for large Hadoop (MapReduce-style) computations, and so application performance will depend on achieving high bisection network bandwidth within the pod. Therefore, you would like to use a multipath-optimized network topology, such as FatTree [4] or HyperX [3].

Each of the basic topologies has numerous parameters, such as the number of ports per switch, and you also must consider the cost of the network: switches, cables, connectors, and the labor to install it. So, how do you decide which topology and parameters are best for your pod? The design space, as we will show, can be complex and not always intuitive.

Now consider the problem from the point of view of a server-pod vendor: how do we decide what network designs to offer customers who expect us to ship them a container with a pre-wired network? The vendor would like to meet the needs of many different kinds of customers (e.g., Hadoop, scientific computing, Web hosting, Cloud computing) without having to support too many different designs, and without having to stock too many different kinds of parts.

In this paper, we expose the challenges of the network design problem for pod clusters, as well as other (non-containerized) data-center networks, and we describe a framework, called *Perseus*,[1] to assist the network designer in making the best choices, given a set of constraints. Our goal is to quickly quantify the costs and benefits of a large set of possible designs, and thus significantly reduce the search space. (Speedy design really does matter; good engineers are overworked.)

While other investigators have tried to analyze the relative merits of topology families (e.g., FatTree, HyperX, or BCube) [1, 13], we know of no prior work that describes how to efficiently search the design space for a given family.

The overall network-design problem gives rise to a variety of optimization sub-problems, many of which we will describe in this paper. Our framework allows us to incorporate optimizers for these problems.

Our contributions in this paper include:

- An overall characterization of the problem of topology optimization for data-center networks on the scale of a single container, or similar-sized clusters.
- A description of the workflow for designing a cost-effective network, given specified performance targets and parts costs.
- Description and solution of several interesting optimization problems, which (to our knowledge) have not previously been discussed in the literature.
- Results, based on semi-plausible parts costs, showing that overall network cost depends on topology parameters in ways that are sometimes hard to predict, reinforcing the need for a tool to explore the design space.

1.1 Problems this paper does not address

The topic of exploring the topology space includes several important problems, such as practical solutions to the flow-routing problem within multipath networks,

[1] Perseus slew the snake-haired Medusa.

or accurate estimates of network costs, or comparing the general merits of HyperX vs. FatTree networks, *that we do not intend to solve in this paper*.

The actual bandwidth attainable in a multipath network depends on how flows are routed, and is likely to be less than the network's bisection bandwidth. Various papers have described computationally-feasible methods for routing in such networks [3, 5]. However, without a realistic traffic model, it is hard to choose routes or predict actual bandwidths; we therefore follow the lead of others (e.g., [13]) in using bisection bandwidth as a crude approximation of a network's useful capacity.

We developed our framework in response to a request from product architects and engineers, who must make cost vs. performance tradeoffs. The engineers we work with have accurate volume-cost estimates for parts, *which we cannot use in this paper, because these costs are subject to non-disclosure agreements*. In this paper, so as to provide an *illustration* of the results of our approach, we use Web-retail prices instead of the true parts costs. Further, such retail prices would be a meaningless basis for comparing between (say) HyperX and FatTree topologies for a 1K-server cluster, because the ratios between these prices do not reflect the ratios between actual costs; therefore, we do not attempt to make such comparisons. (Popa *et al.* [13] have made the attempt.)

We also note that the choice between topology families (FatTree, HyperX, BCube, etc.) depends on other considerations beyond bandwidth and parts costs – e.g., which topologies best support energy proportionality at data-center scale [1], or whether a server-based topology such as BCube is acceptable to customers. This paper does not address those considerations.

Although there are similar design constraints for the network that connects multiple pods into a data-center-wide network, quantitatively this is a significantly different problem, and appears to require qualitatively different networking solutions (e.g., hybrid electrical/optical switch architectures [7]). Therefore, we do not currently attempt to extend our approach to this scale.

2 Defining the problem

Our goal is to help a data-center network designer in a world with too many choices, both for how to design a network, and for how to evaluate a set of designs.

2.1 Points of view

We know of several different types of network designers, with somewhat different points of view: large-scale data-center operators, and external design consultants, with sufficient expertise to design their own networks; moderate-scale data-center operators who prefer to stick to well-understood choices; and system vendors.

Our goal is to serve all of these points of view, although the full flexibility of our approach may be of most interest to a system vendor, who has the most ability to choose the low-level parts. System vendors are increasingly asked, by large customers, to do rack-scale or container-scale pre-integration of servers and networks. Vendor designers have a lot of freedom in terms of the parts they can use; however, they are constrained by the need to choose designs that apply to a large number of customers.

Our work on this problem was, in fact, inspired by a request from pod system designers, who need to know how to design rack-level switches that will be useful for a wide variety of customers (and thus must evaluate network design tradeoffs long before knowing the traffic matrix for any specific customers).

2.2 Design choices

At an abstract level, a data-center network is composed of hosts, switches, and links. We focus our attention on flat Layer-2 designs; the use of IP subnets within data centers complicates some of the design choices and is worth further work.

Recent papers have described a variety of scalable, multi-path Ethernet-based data-center network topologies. These designs use a regular topology (we discuss topologies in Sec. 3) and are typically intended to exploit low-cost, "merchant silicon"-based Ethernet switches.

Generally, these topologies have several free parameters, including the number of end-hosts ("terminals") attached to edge switches (in some designs, all switches are edge switches), the number of ports per switch (the switch "radix"), and link speeds, both of which can vary even within a single network. Switch port count, link speeds, and table sizes all affect overall system cost; we defer detailed discussion of costs until Sec. 4.

For example, some existing merchant-silicon switch products support at least 64 10GbE ports; we expect more and faster ports in the relatively near future.

Each host may have one or more NICs, with per-NIC link speeds of 10 Gbps or even higher. In this paper, we generally ignore the details of end-hosts, including issues such as host virtualization (and therefore the use of virtual switches.)

At a level below the abstraction of switches and links, the network designer must consider more mundane issues, such as cables and connectors. These issues can significantly affect costs (see Sec. 4).

The designer must also consider the physical layout of the equipment. We assume that we will need multiple switches per rack (since most inexpensive switches do not have enough ports for all of the server-NIC downlinks in a rack, as well as for all of the necessary switch-to-switch links in a modern topology). Even so, as we will show, the designer then faces a choice between pack-

ing the racks with as many servers as will fit, or avoiding cables that connect a server in one rack with a switch in another rack.

2.3 Design constraints

Network designers face not just choices, but constraints. These can include:

- **number of connectors**: connectors take up area on the faceplates of switches; as cabling complexity increases, this can become a limiting factor.
- **copper cable length**: Copper cables at multi-gigabit speeds have practical length limits, typically 5–10 meters or less.

There are other similar issues (e.g., the weight of copper cables, the finite capacity of cable plenums, the propensity of massive cable bundles to block airflow and complicate cooling, and limits on the bending radius of all kinds of cables) that we have not yet considered. In addition, switch power consumption is both a major concern and beyond our current ability to model; see [1] for relevant work.

The algorithms that we are developing for Perseus can handle some of these constraints; we currently defer others to future work.

2.4 Evaluation metrics

Given a set of candidate network designs, we must compare them on some bases. There are usually numerous metrics by which to compare two designs, and they seldom agree. Rather than trying to find a single optimal design, our approach is to narrow the design space and then to expose the relative merits of the alternatives to a human designer.

We focus on two primary performance metrics: bandwidth and latency.

We use "bisection bandwidth" as a way to measure the capacity of an entire network design.[2] Following Dally [6], we define "bisection bandwidth" as the "minimum bandwidth over all bisections of the network." In this paper, we do not consider any particular multipath routing scheme, but instead assume that whatever routing scheme is chosen can fully utilize the network hardware and cables.

For reasons of cost, designers must often design networks with non-uniform oversubscription (NUO). Our topology generators can easily create NUO networks, but optimizing these designs and visualizing the results requires us to define a scalar performance metric other than network-wide bisection bandwidth. This is a straightforward change to our algorithms (although it can invalidate some of our heuristics, such as H3 in Sec. 6.3), but space does not permit us to discuss the various possible metrics or how they change the results.

[2]Sometimes we use the related term "oversubscription ratio."

We approximate latency in terms of hop count. Ignoring the considerable latency in end-host stacks, switch hops are still the primary source of delay in an *uncongested* data-center network, and is of great importance to applications such as scientific and financial computing. Characterizing the actual per-switch delay is beyond the scope of this paper.

In addition to performance, a network designer must also consider other metrics, including reliability and power. We discuss some prior work on power in Sec. 9.

Note that the topologies we discuss are all multi-path and hence inherently redundant. One could quantify reliability in terms of the vulnerability of the network to a certain number of randomly-chosen link and/or switch failures, but we are not aware of prior work that describes data-center network failure rates. Some researchers have described their designs as fault-tolerant, but (for example) Mysore *et al.* [11] discuss the reconvergence time of PortLand, but do not quantify its underlying vulnerability. However, Guo *et al.* [9] do show how the aggregate throughput of several topologies, including FatTree and their own BCube design, degrade under random failures.

We believe that it would be quite interesting to understand how to simultaneously optimize the tradeoff between bandwidth, latency, fault tolerance, energy, and cost – but this is beyond our current abilities.

3 Multipath topologies

In recent years, researchers have proposed a wide variety of topologies for data-center networks, all with the goal of providing high bisection bandwidth at low cost. Most of these topologies also require choices to be made for a variety of parameters.

Table 1: Symbols used in this paper

N	total number of servers (or external connections)
R	switch radix (port count)
T	terminals connected to a single switch
S	total number of switches
L	levels of a tree
D	dimensions in a HyperX network
K	link bandwidth
W	total number of links in a network
C	number of top switches in a tree

In this paper, we consider these topology families:

- **FatTree**: Rather than limiting our approach to the three-level k-ary fat tree structures described by Al Fares *et al.* [4], we consider a generalized version of the Clos topologies with parametrized *levels* and *fatness* at each level, which were first defined by Öhring *et al.* [12] as Extended Generalized Fat Trees (EGFTs).

  We recursively define an L level EGFT as follows: A level $L = l$ EGFT connects M_l of $L = l - 1$ level EGFTs with C_l top switches; each top switch has a K_l-wide connection to each of the $l - 1$ level EGFTs. (I.e., K_l is the Link Aggregation (LAG) factor.) A level $L =$

1 EGFT has just one switch ($C_1 = 1$), with M_1 servers directly connected to the switch with $K_1 = 1$ (i.e., unit-bandwidth) links. Note that the level-1 EGFT can be generalized further to consider servers with multiple interfaces. We represent an EGFT as EGFT($L, \vec{M}, \vec{C}, \vec{K}$) where $\vec{M}, \vec{C}, \vec{K}$ are vectors of size L.

Properties: The total number of switches S, number of nodes N, and number of links W in a EGFT($L, \vec{M}, \vec{C}, \vec{K}$) can be computed as follows:

$$\begin{aligned} S &= C_L + M_L(C_{L-1} + \\ &\quad M_{L-1}(...(C_3 + M_3(C_2 + M_2))...)) \\ N &= \prod_{l=1}^{L} M_l \\ W &= C_L M_L K_L + M_L(C_{L-1} M_{L-1} K_{L-1} + \\ &\quad M_{L-1}(...(C_2 M_2 K_2 + M_2(M_1))...)) \end{aligned}$$

If all of the multiple links between two switches can be aggregated into a single cable, then the cable count W' will be:

$$\begin{aligned} W' &= C_L M_L + M_L(C_{L-1} M_{L-1} + \\ &\quad M_{L-1}(...(C_2 M_2 + M_2(M_1))...)) \end{aligned}$$

At a level $l \in [1, L]$ of EGFT($L, \vec{M}, \vec{C}, \vec{K}$), each of the C_l top switches provides $M_l K_l$ bandwidth to all the terminal servers in that sub-fattree. Hence, the oversubscription ratio at level l, referred to as O_l is

$$O_l = \frac{\prod_{i=1}^{l} M_i}{C_l M_l K_l} \quad (1)$$

The oversubscription ratio O of a EGFT($L, \vec{M}, \vec{C}, \vec{K}$) is $O = \max_{l=1}^{L} O_l$. The bisection bandwidth is N/O and the maximum number of hops between any two servers is $2L$.

- **HyperX**: HyperX [3] is an extension of the hypercube and flattened butterfly topologies. Switches are points in a D-dimensional integer lattice, with S_k switches in each dimension $k = 1..D$. The dimensions need not be equal. A switch connects to all other switches that share all but one of its coordinates. (E.g., in a 2-D HyperX, a switch connects to all switches in the same row and in the same column.) The link bandwidths $K_1, ..., K_D$ are assumed to be fixed in each dimension, but can vary across dimensions. At each switch, T ports are assigned to server downlinks.

  We can describe a network as HyperX($D, \vec{S}, \vec{K}, T$), with $\vec{S}$ and $\vec{K}$ as vectors. HyperX($D, \vec{S}, \vec{K}, T$) has $\prod_{k=1}^{D} S_k$ switches, $T.\prod_{k=1}^{D} S_k$ servers, and $(S/2).\sum_{k=1}^{D}[(S_k - 1).K_k]$ links.

In this paper we focus on EGFT and HyperX topologies because they are considered attractive for high bisection bandwidth data-center networks. However, we plan to support other interesting server-to-server topologies such as BCube [9] and CamCube [2], as well as traditional 2- or 3-tier topologies, to allow designers improved flexibility.

4 Cost model

In order to optimize the total cost of a network, we must have a cost model. Some costs are relatively easy to model; these include:

- **Parts costs**: These cover things that a system vendor would buy from other suppliers, such as switches, cables, and connectors.
- **Manufacturing costs**: Given the large physical size of a container-based cluster and the relatively small quantities manufactured, cables for these systems are installed by hand. Sec. 4.2.1 discusses this cost.

Other costs are harder for us to model, especially since they depend on factors beyond the costs to manufacture a specific cluster:

- **Design costs**: A network designer must spend considerable time understanding the requirements for a network, then generating and evaluating specific options. Our approach aims to reduce this cost, while improving the designs produced.

  A vendor of container-based clusters would prefer to deal with a limited number of designs, since each new design requires new Quality Assurance (QA) processes, and each new design must be explained to justifiably skeptical customers.
- **SKU costs**: When a system vendor must deal with a large variety of different parts (often called *Stock-Keeping Units* or SKUs), this creates complexity and generally increases costs. One of our goals, therefore, is to generate network designs that require only a small set of SKUs – generally this means only a few types and lengths of pre-built cables.
- **Cost to reconfigure the network**: Some clusters are born large; others have largeness thrust upon them, later on. A good network design allows for the incremental installation of capacity – for example, one rack of servers at a time – without requiring the re-wiring of the existing network. When such rewiring is required, it should be minimized.
- **Maintenance costs**: Electronics, cables, and connectors do fail. A network design that confuses repair people will lead to higher repair costs and/or more frequent mis-repairs.

In the current version of our framework, we model only the parts costs. Because system vendors and their suppliers are reluctant to reveal their actual volume-purchase parts costs, in this paper we instead use Web-published retail prices as proxies for real costs.

We do not claim that these are the costs that would be paid by a system vendor, but they serve to illustrate many of the important tradeoffs, and we can use them as a plausible input to our topology cost evaluation.

4.1 Switch costs

One can choose from a wide variety of switches for use in a data-center network. Switch costs vary tremendously based on feature set, port count, and performance.

For the sake of simplicity, based on a survey of various switch list prices, we will arbitrarily model switches at $500 per 10GbE port, consistent with the value of $450/port used by Popa *et al.* [13]. (Our tool can easily use table-based inputs for specific switch configurations if they are available.) We also assume that switches are available with exactly the mix of connectors that we would like to use, although these might not currently be off-the-shelf configurations.

We assume that all switches are non-blocking – that is, they do not impose a bandwidth limit beyond what is imposed by their port speeds.

Some researchers have considered the use of merchant-silicon Ethernet switch ASICs, along with the associated supporting parts (CPU, PHYs, power supply and fans, circuit board, etc.) to build low-cost special-purpose switches for data-center networks. This might also seem to be a way to model the cost of higher-radix switches (for example, the Broadcom BCM56840 supports 64 10GbE ports). Farrington *et al.* [8] analyzes the costs of one example of such an approach. However, it turns out to be extremely difficult to estimate the parts costs for such switches; prices for these ASICs are usually non-disclosure information, and it is tricky to estimate the cost of the additional parts (not to mention the cost of engineering, manufacturing, QA, and compliance with standards and regulations). Therefore, in this paper we do not attempt to estimate the costs of this approach, although one could guess that it might not be a lot lower until one is dealing with very large quantities of switches.

We note that some recent data-center network designs, such as CamCube [2], dispense entirely with discrete switch hardware. In such designs, all switching is done within a multi-port server/NIC complex – "each server connects directly to a small set of other services, without using any switches or routers" [2]. We believe our approach could easily be extended to model the costs of switchless networks, by setting the switch cost to zero and including appropriate costs for the required additional NIC ports. It might be harder to model the achievable bandwidth and delay in these networks, since the involvement of NICs or server CPUs at each hop could affect performance in complex ways.

4.2 Cabling costs

High-performance cables are expensive, and can easily amount to a significant fraction of the cost of the servers in a cluster. In Sec. 7.4, we will discuss the problem of optimizing the choice among a number of different cable options. In this section, we discuss cable-cost issues, based on published prices.

There are many options for cabling a high-performance network, and Perseus could easily be extended to cover them, but for this paper we have narrowed our focus to a few likely choices: copper or single-mode fiber, using SFP+ or QSFP+ connectors in either case, with 10GbE connectivity in all cases.[3]

For simplicity of both manufacturing and presentation in this paper, we will assume that any given network design uses only a single connector type. QSFP+ connectors have the benefit of working with either electrical or optical cables, which allows flexibility in cable choice without creating complexity in switch-configuration choice. Fiber QSFP+ cables have the electrical-optical conversion chips integral to the connectors, which adds cost but supports this flexibility. Therefore, we assume the use of single-channel SFP+ cables between servers and switches (and sometimes between switches, for short runs), and quad-channel QSFP+ cables for longer or wider paths between switches. [4]

Table 2: Cable prices (dollars) for various lengths

	Single channel	**Quad channel**		
Length (M)	SFP+ copper	QSFP copper	QSFP+ copper	QSFP+ optical
1	45	55	95	—
2	52	74	—	—
3	66	87	150	390
5	74	116	—	400
10	101	—	—	418
12	117	—	—	—
15	—	—	—	448
20	—	—	—	465
30	—	—	—	508
50	—	—	—	618
100	—	—	—	883

Sources: `http://www.cablesondemand.com/` and `http://www.elpeus.com/`

Table 2 shows cable prices, in dollars, for various lengths of copper and fiber cables certified to run at 10 Gbps. Although these are quantity-50 list prices, not the actual costs to a system vendor, for simplicity we will treat cable costs as equal to cable prices.

One implication of these costs is that a container-sized network might well need to use a mix of copper and optical cables to minimize total cost. This is because copper cables are significantly cheaper than optical cables of the same length; however, quad-channel copper cables cannot support 10GbE over more than 5 meters (SFP+ cables can support longer lengths because they use thicker cables, but consume more connector area as a result). [5] Above 5 meters, we must generally use optical cables.

[3] Popa *et al.* [13] advocate using CAT6 cables, which are cheaper but which require perhaps 3 times more power, and may be harder to use in dense wiring plans.

[4] Some cost-optimal configurations might "waste" channels in these quad-channel cables.

[5] "Active" QSFP copper cables can span 20 meters, but cost almost as much as fiber QSFP+ cables and so we do not consider them.

Thus, the cost-optimal network might depend on finding a topology that exploits short cables as much as possible, which in turn affects the optimization of the physical wiring plan (see Sec. 7.1). On the other hand, physical constraints might force the use of fiber cables even when copper cables would be short enough; see Sec. 7.4.

Based on the prices in Table 2, we can model quad-channel copper cables as costing ca. $16 per meter, plus $20 for each connector. Similarly, we can model quad-channel optical cables as costing ca. $5/meter, plus $188/connector. We model single-channel copper cables at $6/meter, and $20/connector. Custom lengths in small quantities will cost more than high-volume standard lengths, but we are offering these as only crude estimates of costs, and they will suffice for our purposes.

We cannot currently model the extra cost of using more than the minimal number of cable SKUs. Instead, our physical layout algorithm (see Sec.7.1) can either generate networks using custom-length cables or using only the standard cable lengths from Table 2, to illustrate the effect on SKU count, which we quantify in Sec. 8.1.

4.2.1 Cable installation costs

Independent of the parts cost for cables, we also must pay to install them. As far as we know, current manufacturing practices require manual installation of cables between switches.

We estimated cable-installation costs based on a recent experience in a 9-rack, 256-server cluster. A skilled installer can install about 20 intra-rack cables per hour, and about 8 inter-rack cables per hour, although the times depend a lot on cable length and distance between racks. (This experience also points out that the installation rate drops a lot if the installer must deal with cables that are cut too long – finding space for the excess cabling is tricky – so an accurate 3-D model for cable runs could be quite useful. Our algorithm in Sec. 7.1 attempts to model cable lengths as accurately as possible.)

In our area, skilled cable installers can charge ca. $50/hour, so for this paper, we assume a cost of $2.50 for installing an intra-rack cable, and $6.25 for an inter-rack cable. These costs are significantly less than optical-cable costs, but they are not negligible when compared to copper-cable costs.[6] In the long run, we expect cable part costs to decline, but installation labor costs could rise unless there are process improvements that make installation quicker.

5 The topology planning workflow

We can now describe a workflow by which a network designer can explore the space of possible topologies. In brief, the steps of this workflow are: (1) The user specifies the problem, and chooses one or more basic topologies to compare; (2) Perseus generates acceptable candidate topologies, generates optimized wiring plans, estimates overall system cost, and generates a visualization of the resulting space. We describe each of these steps in more detail.

5.1 User-specified inputs

The process of network design starts with a set of goals, constraints and other inputs, which must be provided by the user. These fall into several categories:

System parameters:

- Number of servers to support.
- Server NIC bandwidth (e.g., 1GbE or 10GbE).

While many systems include redundant NICs for fault tolerance, we will consider only non-redundant NICs in this paper. *While our tools can handle a variety of NIC and switch-port bandwidths, for simplicity we assume 10GbE throughout this paper.*

System goals:

- Desired minimum bisection bandwidth, internal to the cluster.
- Desired minimum outgoing bandwidth from the entire cluster. To simplify the rest of the process, we convert this into an equivalent number of "phantom" servers. For example, if the designer wants a 1024-server cluster with 10GbE NICs and 100 Gbps of external bandwidth, we design a network for $1024 + (100/10) = 1034$ server-equivalent ports. This gives the designer freedom to attach external connections to any switch in the cluster.

  This approach to external bandwidth creates some potential inaccuracy in our bandwidth calculations, as we discuss in Sec. 10.
- Desired maximum hop count.
- Desired maximum number of racks.

Parts available:

- Server size, in rack units.
- Switch types: a set of possible switch parts, with port counts and speeds (e.g., "48 1GbE ports + 4 10GbE ports"), and their sizes in rack units.
- Cable types (copper or fiber) and available lengths and bundling factors. We would also like to know a cable's cross-sectional dimensions.

For each kind of part, the user must also supply a cost model. (See Sec. 4 for some example cost models.)

System constraints:

- The maximum number of uplink connectors per switch. (If cables are bundled, a single connector supports multiple logical links.)
- Rack height.
- Desired physical layout of racks: maximum row length, maximum number of rows, and width of aisles

[6] Popa *et al.* [13] argue that labor costs dominate the costs of CAT6 cables.

between rows.
- Plenum cross-section dimensions.

The user must also decide on one or more of the base topology types that the tool supports (currently, just Fat-Tree and HyperX). This choice might depend on considerations that we cannot currently model, such as the expandability of a basic design, or the willingness of customers to accept it.

5.2 Generating candidate topologies

The Perseus tool starts by generating a set of candidate abstract topologies: those that meet the constraints and requirements provided by the user. (If no such candidates exist, the user will have to loosen the requirements and try again.)

This step varies depending on the base topology type, and we describe several appropriate algorithms in Sec. 6.

5.3 Converting abstract to physical wiring plans

After choosing a set of candidate topologies, we must then generate physically valid wiring plans before we can assign costs, since cable costs depend on their lengths. Sec. 7 describes this process in detail, including several topology-specific algorithms. Once we have chosen a physical topology for each candidate abstract topology, we can calculate cable lengths and types.

The least-cost wiring plan might use copper cables for shorter distances, since these could be cheaper than fiber cables of the same lengths. Because plenum space might be a concern, especially for copper cables, we then have to calculate whether the cables will fit. If not, we must replace copper with fiber until everything fits. See Sec. 7.4 for more details.

5.4 Visualization of results

Once we have a list of parts, including specific cable types and lengths, we can easily generate an estimate of the total system cost.

Given the large design space, it would be nice to have a clever multi-dimensional, navigable visualization of the space, including costs, bandwidths, and latencies for a large range of designs.

So far, however, we are using 2-D plots (e.g., network cost vs. bisection bandwidth), with curves for a small variety of parameter settings, as a way to approximate slices of a fancy visualization. Sec. 8 includes some examples. We also have a simple tool that shows how wires are laid out in 2-D space, but for any interesting-sized network, the pictures are too dense to fit into this paper.

6 Generating candidate topologies

In this section, we describe our algorithms for generating candidate topologies of several basic types.

Algorithm 1 Generate HyperX candidate topologies

```
Given:
  N: Number of servers in the pod
  S: Number of switches
  R: Number of ports per switch
  T: Number of terminals per switch
Goal:
  An HyperX config with the minimum cabling cost.
R_h = R − T /* number of HyperX ports per switch */
if R_h ≤ 1 then
  return infeasible /* not enough ports */
/* Initialize the set of candidates with the 1-dim config */
C = {D = 1, S_1 = S}
while C ≠ ∅ do
  config = next_config(C)

  if R_h ≥ (Σ_i(S_i) − D) then
    assign_Ks(config) /* see Sec. 6.2 */
    output_config_details(config)

  C = C ∪ split_dimensions(config)
End
```

6.1 Generating HyperX candidate topologies

When designing an abstract HyperX topology, the designer must evaluate a potentially large space of candidates. Recall that the parameters that characterize this family of topologies are HyperX$(D, \vec{S}, \vec{K}, T)$, where D is the number of dimensions, $\vec{S}$ is the vector of number of switches along each dimension, $\vec{K}$ is the vector of link bandwidths along each dimension, and T is the number of terminals (servers) attached to each switch. (As noted in Sec. 5.1, we currently treat external bandwidth requirements as additional phantom servers.)

The goal of the algorithm described in this section is to generate all of the plausible candidate topologies (based on several constraints), which can then be ranked according to their costs. For the sake of simplicity, we assume that all NICs and server ports have the same unit bandwidth, and that all switches are identical and have the same number of servers attached. (In Sec. 6.1.1 we discuss relaxing the last constraint.)

We state the problem formally: Given N servers (or server-equivalents, to account for external bandwidth), and S switches of radix (port count) R, generate a set of the best feasible HyperX topologies.

Algorithm 1 shows our algorithm in pseudo-code. The first step simply derives the number of ports per switch available for intra-cluster links; we call these the "HyperX ports" to distinguish them from terminal (server and external) ports.

The algorithm then iterates over all possible dimensionalities (values for D) and adds the feasible candidates to the candidate set. For each iteration, we:

- Generate the candidate topologies for this dimensionality D. That is, we generate all possible values of $\vec{S}$ for D.

  For $D = 1$, the only candidate is a linear topology.

For each $D > 1$, we take each of the candidates for $D-1$ and, if possible, split one of its dimensions. For example, a 6x6 2-dimensional lattice can be split into a 6x3x2 lattice, and on the next iteration into a 3x3x2x2 lattice.

- Test the structural feasibility of the candidate; each switch must have enough HyperX ports to connect to all the remaining switches along each dimension.
- For a feasible candidate, find the optimal trunking factor (LAG factor) along each dimension – that is, we generate $\vec{K}$. The designer might prefer LAG factors that are a multiple of the connector and cable width (for example, QSFP connectors that support four-channel cables). A naive approach would require us to examine $\prod_{i=1}^{D}(R-T-i)$ (which, in case of a 5-dimensional HyperX with a 96-port switches, translates to about 3.8 billion) different configurations. Sec. 6.2 presents an $O(R)$ algorithm that derives the optimal trunking factors without exploring this huge space.

Note that when we split solutions from dimension $D-1$ in order to generate new candidates for dimension D, we must include as starting points all of the previous candidates, including the infeasible ones – the progeny of an infeasible candidate might themselves be feasible.

From the feasible candidates, we should then prune out those that require too many connectors to fit on a switch faceplate. We defer this step to future work, although it is fairly simple.

The complexity of Algorithm 1 is determined by the number of unique partitions of the set of prime factors of S; this can be very large, but the algorithm runs fast enough for practical values of S.

Once we have generated the entire set of feasible candidates, we can compute bisection bandwidths (using $\min(S_iK_i)(S/4)$ [3]) , maximum hop counts, and construction costs for each of these.

6.1.1 Optimizing HyperX by adding switches

In the description above, we generate a set of HyperX topologies that exactly match the specific number of switches S. We assume that the designer would like to minimize the number of switches, hence the choice of S. However, the construction in Alg. 1 finds only topologies with exactly the requested number of switches.

Sometimes, adding a few switches might enable much better configurations. For example, suppose the designer specifies a 31-switch network. Since 31 is prime, this forces a single linear design (effectively, a full mesh). However, adding one switch allows a much wider variety of candidates (e.g., 8x4 or 4x4x2), which could make the design feasible with fewer switch ports. Even if the specified number of switches is not prime, it might have inconvenient factors, that could be difficult to satisfy unless the number of ports per switch is quite large – e.g., $S = 94$ would require switches with at least $49+T$ ports, but $S = 95$ would work with $24 + T$-port switches.

We have not yet designed an algorithm to help optimize this parameter choice.

Algorithm 2 Optimal $\vec{K}$ assignment

```
Given:
  D: Number of dimensions of HyperX
  S⃗ = (S_1, S_2, ..., S_D): Size of each dimension of HyperX
  R: Switch radix,
  T: Number of terminals per switch,

Initialize:
  P = R − T
  ∀i ∈ [1, D], K_i = 0
  found=TRUE

while (P > 0) AND (found=TRUE) do
  minSK= min_{i=1}^{D} S_i K_i
  found= FALSE
  for i ∈ [1, D] do
    if (S_i K_i = minSK) AND (P ≥ S_i − 1) then
      K_i = K_i + 1
      P = P − (S_i − 1)
      found=TRUE

return K⃗ = (K_1, K_2, ..., K_D)
```

6.2 Optimal HyperX $\vec{K}$ assignment

The bisection bandwidth of a HyperX$(D, \vec{S}, \vec{K}, T)$ depends both on the topology dimensions $\vec{S}$ and the per-dimension link bandwidth multipliers (LAG factors) $\vec{K}$. Here we show how to optimize $\vec{K}$. This is the same as finding an optimal distribution of each switch's available ports among the different dimensions, such that the bisection bandwidth is maximized.

Given: (i) switches with radix R, of which T ports are used for links to servers and (ii) a HyperX network with D dimensions, with sizes $\vec{S} = (S_1, S_2, ..., S_D)$. Our goal is to distribute the remaining $R-T$ ports of each switch among the D dimensions such that the bisection bandwidth of the topology is maximized. Note that for $HyperX(D, \vec{S}, \vec{K}, T)$, the bisection bandwidth is $\min_{i=1}^{D} S_iK_i$.

Problem: Maximize $\min_{i=1}^{D} S_iK_i$ under the following constraints:

$$\forall i, K_i \in \mathbb{Z} \tag{2}$$

$$\sum_{i=1}^{D}(S_i - 1)K_i \leq R - T \tag{3}$$

Our algorithm for assigning the K_is is shown in Algorithm 2. We first set $K_i = 0$ for all i, and initialize the number of spare ports P to $R-T$. At every step, we consider any dimension i with the minimal S_iK_i product. If enough spare ports are available to increase the bandwidth in that dimension, then we increment K_i by 1. Notice that we reduce the spare ports P by $S_i - 1$, as each switch connects to $S_i - 1$ switches in that dimension. We continue this until we do not have enough spare

ports left to increase the bisection bandwidth.

We have a proof that this algorithm returns the optimal assignment of K_is, but it is too lengthy for this paper.

6.3 Generating Fat Tree topologies

As described in Section 3, we consider Extended Generalized Fat Trees (EGFTs) parametrized with number of levels L, a vector for the number of modules aggregated at each level $\vec{M}$, a vector for the number of top switches at each level $\vec{C}$, and a vector for the lag factor at each level $\vec{K}$. The goal of the algorithm presented here is to generate feasible EGFTs with a given number of switches, each with a fixed radix, and connect a given number of servers.

Construction constraints: Given switches with radix R, a EGFT($L, \vec{M}, \vec{C}, \vec{K}$) can be constructed only if the following constraints hold:

- The top-most switches (level L top switches) should have enough ports to connect to all M_L $L-1$ EGFTs with K_L links.

$$M_L K_L \leq R \tag{4}$$

- At each level $1 \leq l < L$, a top switch should have enough ports to connect to all M_l $l-1$ EGFTs with K_l links, along with the ports to connect to the top switches at the $l+1$ level (we refer to these as "uplinks"). Note that there are C_{l+1} top switches at level $l+1$ with K_{l+1} downlinks, and the C_l top switches should have enough uplink ports to account for those downlinks.

$$1 \leq l < L, M_l K_l + \frac{C_{l+1} K_{l+1}}{C_l} \leq R \tag{5}$$

Finding suitable EGFTs: Given S switches with radix R and N servers, there are various EGFTs possible with different oversubscription ratios, maximum hop-counts, and numbers of cables required. We use a recursive procedure (pseudocode shown in Algorithm 3) to systematically explore the space of possible EGFTs. This $O(L^S)$ algorithm constructs EGFTs bottom-up, and currently it outputs all feasible EGFTs it finds. It is easy to modify this algorithm to generate only those EGFTs with oversubscription below a maximum threshold. For instance, setting this threshold to 1 outputs only the EGFTs with full bisection.

We start the recursion with the following inputs: an empty EGFT=$(0,,,)$, the number of modules NM=N, the number of uplinks at this lowest level NUP=1, and the number of remaining switches RS=S. If number of terminals per switch T is also provided, we run recursion with EGFT=$(1, \langle T \rangle, \langle 1 \rangle, \langle 1 \rangle)$, NM=$\lceil \frac{N}{T} \rceil$, NUP=$R-T$, and RS=$S-$NM.

In each recursive call, we add another level to the existing EGFT, until all modules are aggregated into one single topology (base case for recursion: NM$==1$). Note that at each level $l+1$, this routine systematically explores all possible options for picking the number of lower level modules to aggregate M_{l+1}, the number of top switches to use C_{l+1}, and the bandwidth from each top switch to each module K_{l+1}. We make sure that the constraints in equations 4 and 5 are satisfied as we consider possible values for M_{l+1}, C_{l+1}, and K_{l+1}.

Algorithm 3 EGFTRecurse: Recursive function for constructing EGFTs

```
1:  Input:
2:    EGFT: Current topology (l, M⃗, C⃗, K⃗)
3:    NM: Number of modules at this level
4:    NUP: Number of uplinks available at each module
5:    RS: Remaining switches
6:  Global:
7:    R: Switch radix
8:
9:  /* Base case for recursion */
10: if NM == 1 then
11:   Output EGFT as a possible topology
12:
13: /* For each possible aggregation size */
14: for 2 ≤ M_{l+1} ≤MIN(NM, R) do
15:   /* For each possible number of top switches */
16:   for 1 ≤ C_{l+1} ≤ MIN(NUP, RS/(NM/M_{l+1})) do
17:     /* For each possible K value */
18:     for 1 ≤ K_{l+1} ≤ MIN(R/M_{l+1}, NUP/C_{l+1}) do
19:       EGFTRecurse(
20:           EGFT(l+1, M⃗ ∪ M_{l+1}, C⃗ ∪ C_{l+1}, K⃗ ∪ K_{l+1}),
21:           NM/M_{l+1},
22:           (R − (M_{l+1} ∗ K_{l+1})) ∗ C_{l+1},
23:           RS −(C_{l+1}∗NM/M_{l+1})) ;
```

This recursive exploration can generate a lot of topologies. For example, an execution with $N = 1024, R = 48, T = 32$ results in more than 1 billion possible topologies. However, many of these topologies are clearly inferior, with respect to oversubscription ratios, to other similar topologies. Therefore, we implemented four heuristics to prune the output set:

- **H1**: If all modules are being aggregated (i.e., $M_{l+1} == NM$ in the pseudocode), then it does not make sense to consider all possible values for K_{l+1}. To minimize oversubscription, we need to consider only the maximum possible value for K_{l+1}.
- **H2**: Note that the oversubscription ratio of a topology is the maximum ratio across all levels. So, when building up a EGFT, we do not consider any assignment of M, C, and K that achieves a lower ratio than that has already been imposed by choices at the lower levels.
- **H3**: If all modules at a level can be aggregated into one module, i.e., switch radix R is greater than the number of modules NM at a level, then use the maximum aggregation size instead of trying smaller sizes. Smaller aggregation sizes increase the number of levels, consuming more switches and links without improving bisection bandwidth.
- **H4**: At the top-most level, we use the maximum possible and available top switches that use all uplinks at the next lower level, instead of iterating over all possible values for C.

Table 3: Reduction in search space with heuristics

Parameter			Heuristics used					
N	R	T	None	H1	H2	H3	H4	All
1K	48	16	>1.8B	303.9M	64.3M	315.6K	56.2M	521
1K	48	32	1.0B	61.8M	17.8M	16	13.3M	1
2K	48	16	>1.8B	1.2B	205.5M	471.4M	95.8M	31.0K
2K	48	32	>1.8B	220.1M	64.4M	87.9K	21.5M	521
1K	144	16	>1.9B	>1.5B	184.2M	128	>1.7B	1
1K	144	32	>1.9B	>1.5B	331.8M	112	>1.7B	1
2K	144	16	>1.9B	>1.5B	757.8M	128	>1.7B	1
2K	144	32	>1.8B	>1.5B	>994.5M	112	>1.7B	1
			Values above are sizes of the search space					

Table 3 shows how these heuristics, both independently and together, reduce the search space for several examples of N, R and T – by at least five orders of magnitude, for $N \leq 2048$. We explored up to 4 levels, and terminated incomplete jobs (shown as "$> x$") after 5 hours. Run-times when using all heuristics took less than 200msec, except for one case that took 6 sec.

7 Physical layout of data-center cables

In order to convert an abstract topology into a physical wiring plan, we must know the physical dimensions of the data center, including constraints on where cables can be run between racks.

We use Manhattan routing, rather than trying to route cables on diagonals, which could save some cable costs, but might be infeasible given typical cable plenums.

In some cases, the designer must choose between packing servers and switches as densely as possible into racks (generally a good idea, since POD or data-center floor space is often expensive), or ensuring that all server-to-switch cables stay within the server's rack (which can be useful if racks are shipped pre-wired.) We expose this policy choice, and its consequences (in terms of the number of racks required) to the designer.

We assume the use of standard-size racks (about 24 inches wide, 36 inches deep, and 78 inches tall). We assume that racks are arranged in rows; best practices for cooling call for no space between each rack in a row. Rows are separate either by "cold aisles" or "hot aisles" (i.e., either sources of cool air or sinks of heated air). Several considerations govern the choice of aisle widths [14], but generally the cold aisle must be at least 4 feet wide and the hot aisle at least 3 feet wide. For this paper, we assume that both aisles are 4 feet wide; extending our algorithm to handle mixed widths requires another set of decisions, and is future work.

In modern data centers, network cables do not run under raised floors, because it becomes too painful to trace underfloor cables when working on them. Therefore, inter-rack cables run in ceiling-hung trays above the racks. One tray runs directly above each row of racks, but there are relatively few trays running between rows, because too many cross trays are believed to excessively restrict air flow. We believe that the minimum reasonable separation between cross trays is about two rack widths. (We have not yet done the airflow simulations to validate this assumption.)

We note in passing that if the cables are cut much longer than necessary, the bundles of excess cable can create additional air-flow problems, and can also lead to space problems (not enough room), weight problems (especially for overhead trays), and installation problems (someone has to find a place to put these bundles, and to tie them down).

One other issue to note is that rack dimensions, and rack and tray spacings, are generally given in units of feet and inches, while standard cable lengths are in meters. We suspect this use of incommensurable units could lead to some complications in avoiding excess cable loops.

Algorithm 4 Algorithm for wiring cost computation

```
1: Given:
2:   G_l(V_l, E_l): Logical topology
3:   PMap(v): maps v ∈ V_l to position (x, y, z)
4:   RW: Rack Width
5:   CHTH: Distance: top of the rack to ceiling-hung tray
6:   G_CT: Gap, in racks, between two cross trays
7:   Cost(d): maps cable with length d to its cost
8:
9: Initialize:
10:  CableCost= 0
11:
12: for e = (v_1, v_2) ∈ E_l do
13:   len = 0
14:   (x_1, y_1, z_1) ← PMap(v_1)
15:   (x_2, y_2, z_2) ← PMap(v_2)
16:
17:   /* Add length to pull the two ends of the cable to the side of the
      rack */
18:   len += RW
19:
20:   if x_1 == x_2 AND y_1 == y_2 then
21:     /* both ends are in the same rack */
22:     len += |z_1 − z_2|
23:   else
24:     /* not in the same rack; add length to reach trays */
25:     len += z_1 + z_2 + 2 * CHTH
26:
27:     if x_1 ≠ x_2 AND y_1 mod G_CT > 0 AND y_2 mod
        G_CT > 0 AND y_1/G_CT == y_2/G_CT then
28:       /* Exception: Manhattan distance does not work */
29:       distanceToCrossTray1 = RW * (y_1 mod G_CT + y_2 mod
          G_CT)
30:       distanceToCrossTray2 = RW * (2 * G_CT − (y_1 mod
          G_CT + y_2 mod G_CT))
31:       len += |x_1 − x_2| + MIN(distanceToCrossTray1, dis-
          tanceToCrossTray2) ;
32:     else
33:       len += |x_1 − x_2| + |y_1 − y_2|
34:   CableCost += Cost(len)
35:
36: return CableCost
```

7.1 A general algorithm to create wiring plans

In this section, we describe an algorithm (Algorithm 4, in pseudo-code) to generate a physical wiring plans, including specific cable lengths, from an abstract logical topology. The algorithm also computes the total cost of the cables in the wiring plan, including the server-to-switch cables. The algorithm is generic; it works for all topology types.

The logical topology is provided as a graph $G_l(V_l, E_l)$, where the set of vertices V_l contains both servers and switches of the logical topology and edges E_l represents the links. Also given is a function $T(v)$ that provides the size of node $v \in V_l$ in terms of Rack Units (RU). One RU is 1.75 inches.

We assume that the designer provides the number of racks, their arrangement in rows (X rows with Y racks per row), and their two-dimensional spacing (in particular, the widths of the cold and hot aisles).

Therefore, the wiring problem is to figure out a feasible distribution of the servers and switches in the logical topology to the positions in the racks, such that the cable cost is the lowest. We denote the position of a node that is z RU from the top of the y-th rack in row x as (x, y, z).

Although Algorithm 4 is generic, it depends on a topology-specific function $PMap()$ that assigns servers and switches from the logical topology to a point in three-dimensional space. This is a difficult problem, and we discuss it in detail in Sec. 7.2.

The algorithm also depends on a generic function $Cost(len)$ which, for a cable of length len, determines the appropriate cable type and then computes a cable cost. We discuss this issue in Sec. 7.3.

Note that given two vertices v_1, v_2 and their positions $(x_1, y_1, z_1), (x_2, y_2, z_2)$ the Manhattan distance metric between these two positions is not enough to estimate the length of the cable needed in practice because of several reasons: (i) Switch ports and server NIC interfaces can be located anywhere in the middle of the rack, (ii) For proper airflow, cables are pulled to the side of the rack before they are routed up or down in a rack, (iii) Connections between racks have to run via ceiling-hung cable trays, (iv) Cable trays are a few feet – we assume 24 inches – above racks, (v) There are fewer cross trays (trays across the rows) than the number of racks in a row, to allow sufficient air flow.

To account for (i) and (ii), we add half of the rack width for each end of the link in cable length calculations. To account for (iii) and (iv), we consider the length for reaching the trays above the racks.

We note that even with constraint (v), except in one case, we can use the Manhattan distances between racks to compute the length of cables within the trays themselves. The only exception is when connecting two racks R_i and R_j in different rows, when neither rack is directly under a cross tray, but both R_i and R_j are between a pair of consecutive cross trays. In this case, we pick the cross tray that minimizes the cable length.

7.2 Logical to physical mapping functions

Algorithm 4 imports a topology-specific function $PMap(v)$ that assigns physical space positions to servers and switches in the logical topology ($v \in V_l$). Finding a $PMap(.)$ that minimizes cable cost is a hard problem (we believe it is NP-hard, but we have not yet worked out a reduction). The solution space is huge, because any permutation of nodes in the logical topology is a legal assignment for any subset of rack positions with size $|V_l|$.

We have designed heuristics for $PMap(.)$ for the HyperX and EGFT topology types.

For FatTree networks, we pack servers and switches in order to fill racks as much as possible. For HyperX networks, we chose instead to avoid any cable that runs between a server in one rack and a switch in another; this means that some of our HyperX results might require more racks than strictly necessary. In the future we will modify our algorithms to give the designer the choice between these options.

HyperX: Note that a HyperX topology is symmetric along its dimensions. Also, in HyperX topologies, all switches are edge switches. We leverage these observations to treat all the switches within a dimension as identical, which reduces the search space for rack space assignments. We consider all permutations (orderings) of the dimensions. Each permutation defines the sequence in which switches are assigned to racks. Suppose rh denotes the rack height and eh denotes the height of an edge switch plus the height of the associated servers. Then we can pack $\lfloor rh/eh \rfloor$ edge switches and associated servers into each rack, using the chosen sequences.

We currently assume $eh = 1 + T$ RU (i.e., servers and switches are all 1 RU high); this is probably wrong for high-radix switches, but not significantly wrong.

EGFT: We employ a simple heuristic for packing the nodes of an EGFT. We first pack the edge switches and associated servers in a fashion similar to the one we describe for HyperX, except that we do not have any dimensions in an EGFT on which to randomize. We then iterate from bottom to top and left to right of the logical topology, distributing the switches at each level to the first available space in each of the partially-populated racks. If no such rack is available, then we choose an empty rack. We fill rows before crossing between aisles; this heuristic seems to give shorter cable lengths.

7.3 Cable cost calculation

For the function $Cost(len)$ used in Algorithm 4, we must compute the cost of a cable of length len. We support either of two scenarios: standard-length cables,

based on data such as in Table 2, or custom-length cables.

We first attempt to use a single-channel SFP copper cable for any logical link with LAG factor 1. For wider links, we use quad-channel QSFP copper cables for runs up to 5 meters, or quad-channel QSFP+ fiber cables for longer runs.

When using standard cables, we always choose a cable from the list of standard parts whose length is at least as long as required, and we obtain the cable cost from that list. Sometimes this results in a lot of excess cabling to hide, for longer cable runs.

When using custom cables, our current implementation calculates the cost for a cable of the exact length required.[7] We use the cost model for connectors and cables from Sec. 4.2. We then inflate these costs by an arbitrary 25% to account for the extra costs of purchasing custom cables and carrying them in inventory.

7.4 Choosing between copper and optical

Because high-speed optical network cables cost more than copper cables (see Table 2 for examples), normally we would prefer to use copper cables – if the cable length is below the limit on 10GbE copper cables, approximately 5 meters.

However, if plenum space is limited and we have to route a lot of cables, copper cables might not fit. In this case, we would need to replace some of the copper cables with optical cables.

We have deferred the solution of this problem to future work, especially because (based on some preliminary estimates) it does not appear to be a real problem for moderately large networks.

8 Examples of Perseus results

In this section, we present some results showing the ranges of cost vs. performance tradeoffs that Perseus exposes to the network designer. *(Remember, these results are based on only semi-plausible parts costs, and should never be quoted as realistic network costs.)*

For reasons of space, we limit the results presented here to configurations with N (servers) set to either 1024 or 8192. We consider a variety of switch radices: 32, 48, 64, 96, and 144, and we consider T (servers/switch) values of 16, 24, and 32. For HyperX networks, we explored designs with "excess" bisection bandwidth, since HyperX is not an "equal-cost multipath" topology, and optimal routing could be difficult.

Figures 1 and 2 show cost vs. bandwidth curves for HyperX and FatTree configurations for, respectively, 1K and 8K servers. Cost is based on our models for switch cost and for standard-length cables, including installation labor, and includes server-to-switch cables. For the HyperX curves, we plot curves for just a few values of S (number of switches) to preserve readability, and the points on each curve reflect various choices for D and $\vec{S}$. For the FatTree curves, the points on each curve reflect various choices for S, but we plot curves only for selected values of T, to keep the graphs readable. In all cases, we plot only the least-cost configuration that achieves a target bisection bandwidth. The curves show only the points for the least-cost physical layout for a given abstract topology. Note that naive designs could have much higher costs than the designs we generate.

Figures 1 and 2 lead to several observations. First, total network cost generally increases with increasing bandwidth, but not always; especially for larger HyperX networks, one can often find a "better" configuration at a much lower cost through a small parameter changes. Second, for a given target bandwidth, there are often several possible parameter choices with widely varying costs. Finally, although the figures suggest that FatTree networks might be somewhat less expensive than HyperX networks for the same target bandwidth, our cost models are currently too crude to support this as a general conclusion.

Total costs obviously depend on our models for parts costs; Fig. 3 shows how the HyperX($N = 8192$) results would change if switches cost just 20% as much as in Fig. 2 (the same low-cost model as in [13]). Note that the relative ranking of the curves usually does not change, but some of the "sweet spots" do.

Figure 4 shows how cost varies with HyperX network worst-case hop counts. (For FatTree networks, the worst-case hop count is simply $2L$; L=6 for all of the configurations plotted in Fig. 2.) One can sometimes, by spending more money, reduce the worst case hop count by one or two, but generally high-bandwidth topologies also have low hop counts.

Computation time: Computation costs are tolerable, especially since generating these graphs parallelizes easily. Table 4 shows elapsed times for various N.

Table 4: Computation costs (wall times)

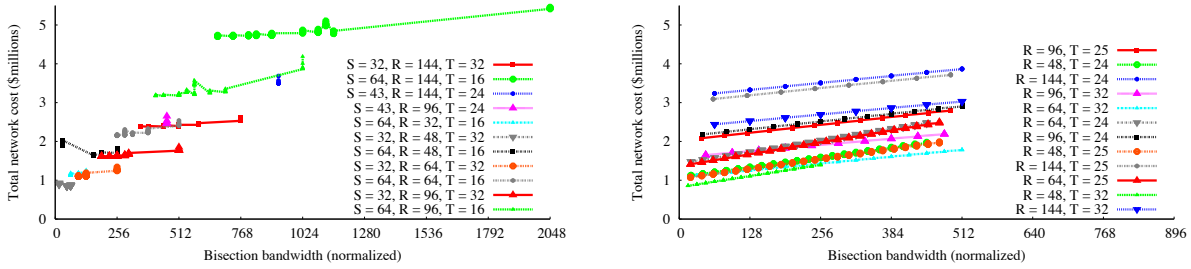
Network	N	Xeon CPUs	Jobs	Worst-case	Total
HyperX	8K	3GHz	166	155s	3.4hr
HyperX	16K	3GHz	68	20min	14.6hr
FatTree	8K	2.33GHz	96	55min	28.8hr

8.1 Secondary metrics

We found that using custom cables does not increase costs very much, based on our crude model for their parts cost. (Space does not permit us to show these graphs.)

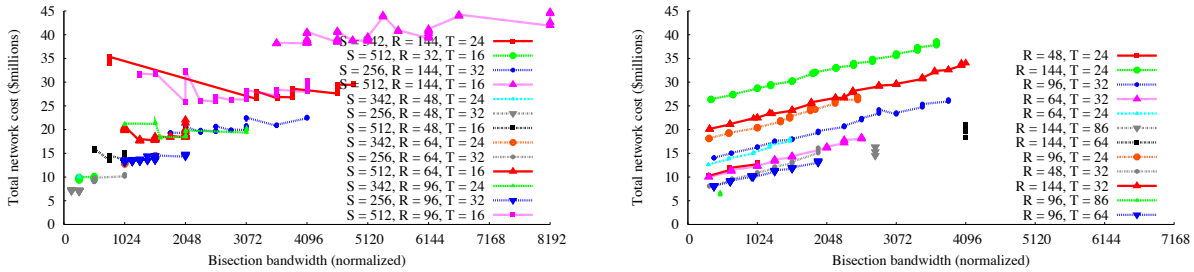
However, custom cables do significantly reduce the amount of excess cable that must be tucked away without blocking airflow. For example, for HyperX

[7] We plan to modify this so that it quantizes the lengths in multiples of perhaps 0.5 meters, thus reducing SKU count at the cost of having to hide some slight excess cabling. This would also allow us to use standard cables if they are available in the right length.



HyperX configurations FatTree configurations

Figure 1: Cost vs. bisection bandwidth for 1024-server networks

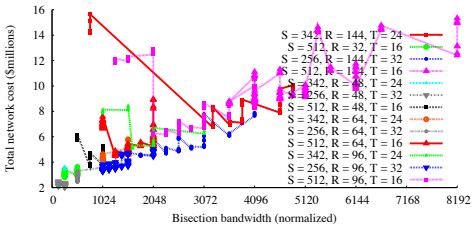


HyperX configurations FatTree configurations

Figure 2: Cost vs. bisection bandwidth for 8192-server networks

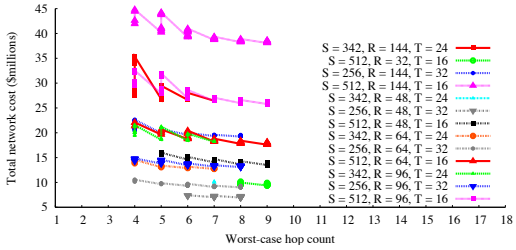
with N=8192, the least-cost full-bandwidth configuration with standard cables requires 9 SKUs, or 141 SKUs using custom cables. When using standard cables, HyperX/N=8192 configurations end up with mean per-cable excess lengths of between 23 and 265 in.

We also can quantify the number of wasted lanes in quad-channel cables. For HyperX with N=8192, this ranges from 0 to >55K, depending on the configuration.



HyperX configurations, $N = 8192$

Figure 3: Cost vs. BW for $100 switch ports



HyperX, $N = 8192$

Figure 4: Cost vs. worst-case hop count

9 Related work

Traditional topology analyses did not focus on physical layout issues that arise in a data center; they mostly considered only logical metrics such as bisection bandwidth. Much of the prior work also does not address the problem of efficiently sweeping through the logical topology space [6].

Ahn *et al.* [3] presented an algorithm that minimizes the number of switches needed to build a HyperX network with a given bisection. However, they do not consider any layout issues.

Popa *et al.* [13] compared the costs of several datacenter network architecture families. Their analysis covered the use of server cores for switching, and included energy costs. However, they did not directly address the problem of finding a least-cost network design within a specific topology family, and they do not appear to have tried to optimize the placement of switches in racks.

Farrington *et al.* [8] analyzed cabling issues for FatTree networks. They showed that much cost could be eliminated by consolidating the upper levels of a FatTree, replacing cables, connectors, and a physically distributed set of low-radix commodity switches with a design using merchant silicon. Many of the cables become traces on circuit boards. They also show how to use higher-speed links within a FatTree to reduce the cable count.

The Helios [8] design also addresses the costs of cabling, switching, and especially the costs of electrical to optical conversions. Helios focuses on the connec-

tions between containers, and exploits low-cost, relatively slow optical switches, so it covers a somewhat different domain than Perseus is aimed at.

Currently our tools do not model topology related energy costs. At one level, quantifying power is easy: we simply add up the power consumption of the individual switches. However, as switch designers improve the energy-proportionality of their products, switch power becomes more dependent on traffic demands (that is the goal of proportionality). An accurate estimate of network-related power consumption requires a detailed, time-varying traffic model, and also requires accurate models for switch power consumption at various traffic levels. These data are often hard to obtain, which could make it difficult to compare topology-related energy costs; Popa *et al.* [13] used average network-wide utilization as a proxy. Abts *et al.* [1] demonstrated that a flattened butterfly topology is more power-efficient than a folded-Clos (FatTree) network.

Prior work by Heller *et al.* [10] has shown that dynamic adjustment of the set of active, power-consuming links can increase network-wide proportionality. Thus, there is a complex interaction between network topology, traffic demands, and power consumption, and we do not know how to model this issue in detail.

10 Future work

Within our framework, there is a lot of room for further work, including:

- Understanding how to model internal vs. external bandwidth. Currently, we assume that local servers and external connections properly share the overall bisection bandwidth of a network, but this is probably wrong. We also need to understand whether designers care where the external ports can be connected.
- Dealing different widths for hot and cold aisles.
- Incorporating plenum capacity into copper vs. fiber choice and/or design-feasibility testing (See Sec. 7.4).
- Modeling network energy consumption. This requires models for traffic and switch energy proportionality.
- Modeling network reliability.

11 Summary and conclusions

In this paper we have exposed the complexity of the problem of choosing good designs for high-bandwidth, multi-path data-center networks. We have described the Perseus framework for assisting network designers in exploring this space, and we have presented several algorithms that help to optimize parameter choices. We have also shown that, based on semi-plausible parts costs, the overall cost for constructing a network with a given bisection bandwidth can vary significantly.

We know from past experience that the relative costs of parts such as switches and cables will change over time, sometimes dramatically. We also expect NIC and switch port speeds to increase in several jumps. These trends mean that topology choices based on current parts will undoubtedly need to be re-evaluated every year or two; thus, our goal in this paper has been to provide a methodology and a set of algorithms, not to describe the "correct" choices for topologies and their parameters.

While designers of real networks will undoubtedly use different costs, they will still have to grapple with the choice of parameters, and Perseus should prove useful in this task.

References

[1] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy Proportional Datacenter Networks. In *ISCA*, pages 338–347, 2010.

[2] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic Routing in Future Data Centers. In *SIGCOMM*, 2010.

[3] J. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *Proc. Supercomputing*, Nov. 2009.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[5] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proc. NSDI*, Apr. 2010.

[6] W. J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufman, 2004.

[7] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM*, 2010.

[8] N. Farrington, E. Rubow, and A. Vahdat. Data Center Switch Architecture in the Age of Merchant Silicon. In *Proc. Hot Interconnects*, pages 93–102, 2009.

[9] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proc. SIGCOMM*, Barcelona, 2009.

[10] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: Saving energy in data center networks. In *Proc. NSDI*, 2010.

[11] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proc. SIGCOMM*, 2009.

[12] S. Öhring, M. Ibel, S. Das, and M. Kumar. On generalized fat trees. In *Proc. Parallel Proc. Symp.*, 1995.

[13] L. Popa, S. Ratnasamy, G. Iannaccone, A. Krishnamurthy, and I. Stoica. A Cost Comparison of Data Center Network Architectures. In *Proc. CoNEXT*, Dec. 2010.

[14] N. Rasmussen and W. Torell. Data Center Projects: Establishing a Floor Plan. White Paper 144, American Power Conversion, 2007.

In-situ MapReduce for Log Processing

Dionysios Logothetis, Chris Trezzo*, Kevin C. Webb, and Kenneth Yocum
*UCSD Department of Computer Science, *Salesforce.com, Inc.*

Abstract

Log analytics are a bedrock component of running many of today's Internet sites. Application and click logs form the basis for tracking and analyzing customer behaviors and preferences, and they form the basic inputs to ad-targeting algorithms. Logs are also critical for performance and security monitoring, debugging, and optimizing the large compute infrastructures that make up the compute "cloud", thousands of machines spanning multiple data centers. With current log generation rates on the order of 1–10 MB/s per machine, a single data center can create tens of TBs of log data a day.

While bulk data processing has proven to be an essential tool for log processing, current practice transfers all logs to a centralized compute cluster. This not only consumes large amounts of network and disk bandwidth, but also delays the completion of time-sensitive analytics. We present an in-situ MapReduce architecture that mines data "on location", bypassing the cost and wait time of this store-first-query-later approach. Unlike current approaches, our architecture explicitly supports reduced data fidelity, allowing users to annotate queries with latency and fidelity requirements. This approach fills an important gap in current bulk processing systems, allowing users to trade potential decreases in data fidelity for improved response times or reduced load on end systems. We report on the design and implementation of our in-situ MapReduce architecture, and illustrate how it improves our ability to accommodate increasing log generation rates.

1 Introduction

Scalable log processing is a crucial facility for running large-scale Internet sites and services. Internet firms process click logs to provide high-fidelity ad targeting, system and network logs to determine system health, and application logs to ascertain delivered service quality. For instance, E-commerce and credit card companies analyze point-of-sales transactions for fraud detection, while infrastructure providers use log data to detect hardware misconfigurations and load-balance across data centers [6, 30].

This semi-structured log data is produced across one or more data centers that contain thousands of machines. It is not uncommon for such machines to produce data at rates of 1–10 MB/s [4]. Even at the low end (1 MB/s), a modest 1000-node cluster could generate 86 TB of raw logs in a single day. To handle these large data sets, many sites use data parallel processing systems like MapReduce [12] or Dryad [20]. Such frameworks allow businesses to capitalize on cheap hardware, harnessing thousands of commodity machines to process enormous data sets.

The dominant approach is to move the data to a single cluster dedicated to running such a bulk processing system. In this "store-first-query-later" approach [13] users load data into a distributed file system and then execute queries.[1] For example, companies like Facebook and Rackspace analyze tens of terabytes of log data a day by pulling the data from hundreds to thousands of machines, loading it into HDFS (the Hadoop Distributed File System), and then running a variety of MapReduce jobs on a large Hadoop cluster [17]. Many of the processing jobs are time sensitive, with sites needing to process logs in 24 hours or less, enabling accurate user activity models for re-targeting advertisements, fast social network site updates, or up-to-date mail spam and usage statistics.

However, this centralized approach to log processing has two drawbacks. First, it fundamentally limits its scale and timeliness. For example, to sink 86 TB of log data in less than an hour (48 minutes) would require 300 Gb/s of dedicated network and disk bandwidth. This limits its processing on the MapReduce cluster as the transfer occupies disk arms, and places a large burden on the data

[1] Here we consider queries as single or related MapReduce jobs.

center network, even if well provisioned. Second, the approach must sacrifice availability or blindly return incomplete results in the presence of heavy server load or failures. Current bulk processing systems provide strict consistency, failing if not all data is processed. This implies that either users delay processing until logs are completely delivered or that their analytics run on incomplete data.

In fact, though, one does not have to make this either-or choice. It is often possible to accurately summarize or extract useful information from a subset of log data, as long as we have a systematic method for characterizing data fidelity. For example, if a user can ascertain whether a particular subset of log data is a uniform sampling, one can capture the relative frequency of events (e.g., failures or user clicks) across server logs.

To meet these goals we present an "in-situ" MapReduce (iMR) architecture for moving analytics on to the log servers themselves. By transforming the data in place, we can reduce the volume of data crossing the network and the time to transform and load the data into stable distributed storage. However, this processing environment differs significantly from a dedicated Hadoop cluster. Nodes are not assumed to share a distributed file system, implying that data is not replicated nor available at other nodes. And the servers are not dedicated to log processing; they must also support client-facing requests (web front ends, application servers, databases, etc.). Thus unlike traditional MapReduce architectures, our in-situ approach accepts that data may naturally be unavailable either because of failures or because there are insufficient resources to meet latency requirements.

This work makes the following contributions:

- **Continuous MapReduce model:** Unlike batch-oriented workloads, log analytics take as input essentially infinite input streams. iMR supports an extended MapReduce programming model that allows users to define continuous MapReduce jobs with sliding/tumbling windows [7]. This allows incremental updates, re-using prior computation when data arrives/departs. Because iMR directly supports stream processing, it can run standard MR jobs continuously without modification.
- **Lossy MapReduce processing:** iMR supports lossy MapReduce processing to increase result availability when sourcing logs from thousands of servers. To interpret partial results, we present C^2, a metric of result quality that takes into account the spatial and temporal nature of log processing. In iMR users may set a target C^2 for acceptable result fidelity, allowing the system to process a subset of the data to decrease latency, avoid excessive load on the log servers, or accommodate node or network failures.
- **Architectural lessons:** We explore the iMR architecture with a prototype system based on a best-effort distributed stream processor, Mortar [22]. We develop efficient strategies for internally grouping key-value pairs in the network using sub-windows or *panes*, and explore the impact of failures on result fidelity and latency. We also develop load cancellation and shedding policies that allow iMR to maximize result quality when there are insufficient server resources to provide perfect results.

Section 2 gives an overview of the system design, discusses related work, and describes how iMR performs continuous MapReduce processing using windows. Section 3 introduces our notion of result quality C^2, useful ways to express C^2, and how the system efficiently maintains that metric. Section 4 discusses our modifications to Mortar to support iMR. We evaluate the system in Section 5, looking at system scalability, load shedding, and data fidelity control. In particular we explore how C^2 affects results when extracting simple count statistics, performing click-stream analysis, and building an HDFS anomaly detector.

2 Design overview

iMR is designed to complement, not replace traditional cluster-based architectures. It is meant for jobs that filter or transform log data either for immediate use or before loading it into a distributed storage system (e.g., HDFS) for follow-on analysis. Moreover, today's batch processing queries exhibit characteristics that make them amenable to continuous, in-network processing. For instance, many analytics are highly selective. A 3-month trace from a Microsoft large-scale data processing system showed that filters were often highly selective (17 - 26%) [16], and the first step for many Facebook log analytics is to reduce the log data by 80% [4]. Additionally, many of these queries are update-driven, integrate the most recent data arrivals, and recur on an hourly, daily, or weekly basis.

Below we summarize how in-situ MapReduce ensures that log processing is:

Scalable: The target operating environment consists of thousands of servers in one or more data centers, each producing KBs to MBs of log data per second. In iMR, MapReduce jobs run continuously on the servers themselves (shown on the right in Figure 1). This provides horizontal scaling by simply running in-place, i.e, the processing node count is proportional to the number of data sources. This design also lowers the cost and latency of loading data into a storage cluster by filtering data on site and using in-network aggregation, if the user's reduce implements an aggregate function [14].

Responsive: Today the latency of log analytics dictates various aspects of a site's performance, such as the

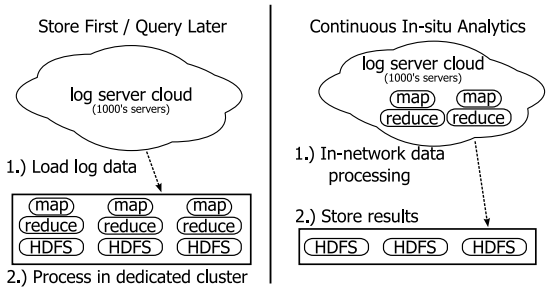


Figure 1: The in-situ MapReduce architecture avoids the cost and latency of the store-first-query-later design by moving processing onto the data sources.

speed of social network updates or accuracy of ad targeting. The in-situ MapReduce (iMR) architecture builds on previous work in stream processing [5, 7, 9] to support low-latency continuous log processing. Like stream processors, iMR MapReduce jobs can process over sliding windows, updating and delivering results as new data arrives.

Available: iMR's lossy data model allows the system to return results that may be incomplete. This allows the system to improve result availability in the event of failures or processing and network delays. Additionally, iMR may pro-actively reduce processing fidelity through load shedding, reducing the impact on existing server tasks. iMR attaches a metric of result quality to each output, allowing users to judge the relative accuracy of processing. Users may also explicitly trade fidelity for improved result latency by specifying latency and fidelity bounds on their queries.

Efficient: A log processing architecture should make parsimonious use of computational and network resources. iMR explores the use of sub-windows or *panes* for efficient continuous processing. Instead of recomputing each window from scratch, iMR allows incremental processing, merging recent data with previously computed panes to create the next result. And adaptive load-shedding policies ensure that nodes use compute cycles for results that meet latency requirements.

Compatible: iMR supports the traditional MapReduce API, making it trivial to "port" existing MapReduce jobs to run in-situ. It provides a single extension, *uncombine*, to allow users to further optimize incremental processing in some contexts (Section 2.3.2).

2.1 In-situ MapReduce jobs

A MapReduce job in iMR is nearly identical to that in traditional MapReduce architectures [12]. Programmers specify two data processing functions: map and reduce. The map function outputs key-value pairs, $\{k, v\}$, for each input record, and the reduce processes each group of values, $v[]$, that share the same key k. iMR is designed for queries that are either highly selective or employ reduce functions that are distributive or algebraic aggregates [14]. Thus we expect that users will also specify the MapReduce *combiner*, allowing the underlying system to merge values of a single key to reduce data movement and distribute processing overhead. The use of a combiner allows iMR to process windows incrementally and further reduce data volumes through in-network aggregation. The only non-standard (but optional) function iMR MapReduce jobs may implement is *uncombine*, which we describe in Section 2.3.2.

However, the primary way in which iMR jobs differ is that they emit a stream of results computed over continuous input, e.g., server log files. Like data stream processors [7], iMR bounds computation over these (perhaps infinite) data streams by processing over a *window* of data. The window's *range* R defines the amount of data processed in each result, while the window's *slide* S defines its update frequency. For example, a user could count error events over the last 24 hours of log records ($R = 24\,\texttt{hours}$), and update the count every hour ($S = 1\,\texttt{hour}$). This *sliding* window, one whose slide S is less than its range R, may be in terms of wall-clock time or logical index, such as record count, bytes, or any user-defined sequence number. Users specify R and S with simple annotations to the reduce function.

While sufficient for real-time log processing, a MapReduce job in iMR may reference historical log data as well. Doing so requires a job-level annotation that specifies the point in the local log to *begin* B and the total data to consume, the *extent* E. If unspecified, the job continues to process, possibly catching up to real-time processing.

2.2 Job execution

In general, MapReduce architectures have three primary tasks: the parallel execution of the map phase, grouping input records by key, and the parallel execution of the reduce phase. In cluster-based MapReduce systems, like Hadoop, each map task produces key-value pairs, $\{k,v\}$, from raw input records at individual nodes in the cluster. The map tasks then group values by their key k, and split the set of keys into r partitions. After the map tasks finish, the system starts a reduce task for each partition r. These tasks first download their partition's key-value pairs from each mapper (the *shuffle*), finish grouping values, and then call reduce once for every $\{k,v[]\}$ pair.

iMR distributes the work of a MapReduce job across multiple trees, one for each reducer partition. Figure 2 illustrates one such tree; iMR co-locates map processing on the server nodes themselves, sourcing input records

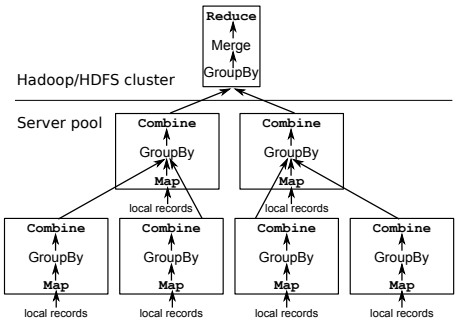


Figure 2: This illustrates the physical instantiation of one iMR MapReduce partition as a multi-level aggregation tree.

(tuples) from the local node's log file. The dedicated processing cluster hosts the root, which executes the user's reduce function. This tree uses the combine API to aggregate intermediate data at every mapper in a manner similar to traditional MapReduce architectures. However, like Dryad [32], iMR can use multi-level aggregation trees to further reduce the data crossing the network.

In general, this requires aggregate or *decomposable* functions that can be computed incrementally [15, 23, 32]. Here we are interested in two broad categories of aggregate functions [21]. *Holistic* aggregates require partial values whose size is in proportion to their input data, e.g., `union`, `median` or `groupby`. In contrast, *bounded* aggregates have constant-sized partial values, e.g., `sum` or `max`, and present the greatest opportunities for data reduction.

2.3 Window processing with panes

iMR supports sliding processing windows not just because they bound computation on infinite streams, but because they also enable incremental computations. However, they do not immediately lend themselves to efficient in-network processing. Consider a simple aggregation strategy where each log server accumulates all key-value pairs for each logical window and nodes in the aggregation tree combine these entire windows.

We can see that this strategy isn't efficient for our example sliding window query. In this case, every event record would be included in 24 successive results. Thus every input key-value pair in a sliding window would be grouped, combined, and transmitted for each update (slide) of the window or R/S times. To reduce these overheads, iMR adapts the use of sub-windows or *panes* to efficiently compute aggregates over sliding windows. While the concept of panes was introduced in prior work for single-node stream processors [21]; here we adapt them to distributed in-situ MapReduce processing.

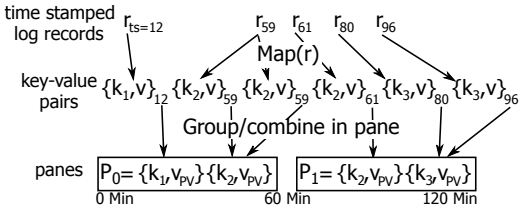


Figure 3: iMR nodes process local log files to produce sub-windows or panes. The system assumes log records have a logical timestamp and arrive in order.

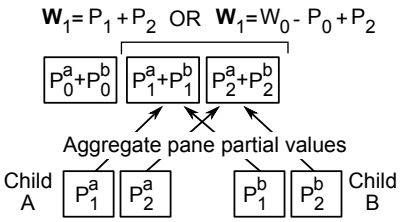


Figure 4: iMR aggregates individual panes P_i in the network. To produce a result, the root may either combine the constituent panes or update the prior window by removing an expired pane and adding the most recent.

2.3.1 Pane management

Panes break a window into multiple equal-sized sub-windows, allowing the system to group and combine key-value records once per sub-window. Nodes in the system generate panes and send them to their parents in the aggregation tree. Thus in iMR, interior nodes in a tree aggregate panes and the root node combines them into each window result. This supports the fundamental grouping operation underlying reduce, a holistic aggregate. By sending panes, rather than sending the entire window up the tree, the system sends a single copy of a key's value, reducing network traffic. Additionally, issuing values at the granularity of panes gives the system fine-grain control on fidelity and load shedding (Section 3.4). It is also the granularity at which failed nodes restart processing, minimizing the gap of dropped data (Section 4.4.2).

Figure 3 illustrates how a single node creates panes from a stream of local log records. Typically, we set the pane size equal to the slide S, though it may be any common divisor of R and S, and each node maintains a sequence of pane partial values P_i. This example uses a processing window with a slide of 60 minutes. When log records first enter the system, iMR tags each one with a non-decreasing user-defined timestamp. The system then feeds these records to the user's map function. After mapping, the system assigns key-value pairs

to each pane, where they are grouped/combined. Note that a pane is complete when a log entry arrives for the following pane (log entries are assumed to be in order).

2.3.2 Window creation

In iMR, the root of each reducer partition must group/combine all keys in the window before executing the user's reduce function and computing the result. Figure 4 illustrates two strategies the root may employ to do so. Here two log servers A and B create panes P_1 and P_2 and send them to the root. The root first groups (and possibly combines) panes with the same index.

The first strategy leverages panes to allow incremental processing with the traditional MapReduce API. The strategy simply uses the existing combine API to merge adjoining panes. In this example each window consists of two panes and W_1 may be constructed by merging P_1^{a+b} with P_2^{a+b}. This improves efficiency by having each overlapping window re-use a pane's partial value; merging window panes is cheaper than repeatedly combining the raw mapped values for each window. This benefit increases with the number of values per key.

However, for sliding windows it is sometimes more efficient to *remove* expired data and then add new data to the prior W. For instance, consider our 24 hour query that updates every hour. In this case the root must combine 24 panes to produce each window. In contrast, the root could remove and add a pane's worth of keys to the prior window W, greatly reducing the volume of keys touched. Assuming that the cost of removing and adding keys to W is equivalent, this strategy is always more efficient than merging all constituent panes when the slide is less than half the range. This requires "differential" [21] functions, i.e. aggregates that are commutative/associative under removals as well as additions. iMR only uses an uncombine strategy when the slide is less than half the range and a user supplies an uncombiner.

3 Lossy MapReduce processing

This section describes the features of iMR that allow it to accommodate data loss. As described earlier, data loss may occur because of node or network failures, or as a consequence of result latency requirements. In such cases, an iMR job may need to report a result before the system has had time to process all the data in the window. The key challenges we address here are a.) how to represent and calculate result quality to allow users to interpret partial results, and b.) how to use this metric to trade result fidelity for improved result latency.

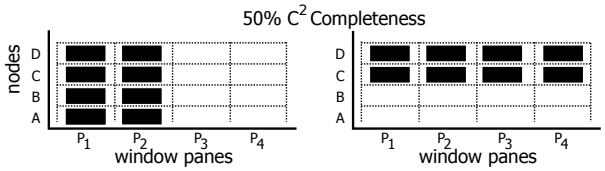


Figure 5: C^2 completeness describes the set of panes each log server contributes to the window. Here we show two different ways in which C^2 represents 50% of the data area: all the nodes process half the data or half the nodes process all the data.

3.1 Measuring data fidelity

A good measure of data fidelity should inform users not only that data is missing, but allows users to ascertain the impact of data loss on query accuracy. One measure of result quality used for in-network aggregates is *completeness*, the number or fraction of nodes whose data is represented in the final answer [22, 25]. Alternatively, systems like Hadoop Online (HOP) output partial answers as data arrives, and annotate them with *progress*, the percent of total data processed. Unfortunately, neither metric is sufficiently descriptive for window-based processing. Completeness cannot differentiate between a single node that produces log records that span the entire window and a node that does not. Similarly, a simple progress metric fails to account for the source of processed data.

Here we present a completeness metric, C^2, that leverages the natural distribution of log data across both space (log server nodes) and time (the window range). C^2 represents the data *area* included in the final result as the number of unique data panes that have been successfully integrated into the window. Logically, the root maintains C^2 like a scoreboard, with a mark for every successfully received pane in the window (Figure 5). Thus C^2 tracks the set of nodes whose log data contributed to the window, as well as how that log data was distributed across the result window. iMR can summarize this raw information as independent percentages of temporal (x-axis) and spatial (y-axis) completeness or simply as an area, the total result coverage.

Figure 5 illustrates how C^2 may reflect two different scenarios that process the same data area (in this case C^2 =50%). In the first case, all the nodes process half the data and in the second, half the nodes process all the data. There are, of course, other scenarios where the product of the percentage of nodes and percentage of the window processed will be 50%, and C^2 allows users to differentiate between them. Note that C^2 explains what was included in the result, not what was *missing*, which is a much harder (and often query specific) metric to provide.

To measure fidelity, interior nodes aggregate C^2 for

individual panes as they make their way up each reduce tree. Since each pane is by definition temporally complete, representing data for that portion of the window, this per-pane C^2 simply maintains a count and the IDs of data sources summarized in a particular pane. As panes are merged in the aggregation tree, so too is their C^2 information. The root represents C^2 as a histogram with a bin per pane that counts the nodes that responded for that pane. This allows the root to summarize C^2 as the percent of nodes reporting (unique nodes responding divided by total nodes) and the percent window computed (non-empty panes divided by total panes per window).

3.2 Using C^2: applications

This section examines how applications use C^2 to bound result quality and to understand imperfect results. Users specify minimum fidelity requirements by annotating queries with a target fidelity that constrains results along particular spatial and temporal dimensions. For example, applications may specify C^2 as a minimum area A, giving the system a large degree of freedom to meet fidelity requirements, as any set of panes will do. Or applications may specify C^2 as percentages of temporal and spatial completeness: $(\%time, \%space)$. For example, one could require panes in the window to be 100% spatially complete (as they are in the left-hand of Figure 5), but relax the requirement for the other axis.

The goal for an application is to set a fidelity bound that allows users to determine result quality from C^2. In particular, they should fix the axes along which result quality is unpredictable. Thus two results may both meet the fidelity bound, but users can ascertain relative result quality by comparing how they did so. To illustrate these concepts, we now describe four general C^2 specifications and their fidelity/latency tradeoffs.

Area (A) with earliest results: This C^2 specification gives the system the most freedom to decrease result latency (or shed load). Without failure or load shedding, iMR will return the first $A\%$ panes from each log server for the result window. These results will correctly summarize event frequencies only if events were uniformly distributed across the log servers. This is the case with simple applications, such as Word Count, where an approximate answer could be used to estimate the relative frequency of words. However, if some words (events) are associated with some servers more than other words, the data will be biased.

Area (A) with random sampling: This C^2 specification gives the system less freedom to decrease result latency, but tries to ensure that a partial result correctly reproduces the relative occurrence of events in the result window, no matter how events are distributed across the log servers. Here each iMR node randomly creates panes with a probability in proportion to A. This takes longer to reach the fidelity bound than the first strategy, but will correctly sample the log data. Note applications must check the C^2 score to verify a sufficient sample in the event of pane loss due to node or network failures.

Spatial completeness $(X, 100\%)$: This specification ensures that each pane in the result window contains data from 100% of the nodes in the system. It is useful for applications that must correlate log events on different servers that occur close in time. For example, consider a basic click-stream analysis that allows web sites to characterize user behavior. With load-balanced web and application serving architectures, a user's click events may arrive at any log server. Intuitively this C^2 specification captures a spatial "slice" of the log data, collecting a snapshot of user activity across the servers during a pane.

Temporal completeness $(100\%, Y)$: This specification ensures that Y percent of the nodes in the system respond with 100% of the panes in the result window. It is useful for applications that must correlate log events on the same server across time. For example, if in the click-stream analysis, individual users had been assigned/pinned to particular servers, this would be the C^2 to employ.

3.3 Result eviction: trading fidelity for availability

iMR allows users to specify latency and fidelity bounds on continuous MapReduce queries. Here we describe the policies that determine when the root evicts results. The root has final authority to evict a window and it uses the window's completeness, C^2, and latency to determine eviction. Thus a latency-only eviction policy may return incomplete results to meet the deadline, while a fidelity-only policy will evict when the results meet the quality requirement.

Latency eviction: A query's latency bound determines the maximum amount of time the system spends computing each successive window. If the timeout period expires, the operator evicts the window regardless of C^2. Before the timeout, the root may evict early under three conditions: if the window is complete before the timeout, if it meets the optional fidelity bound C^2, or if the system can deduce that further delays will not improve fidelity. Like the root, interior nodes also evict based on the user's latency deadline, but may do so before the deadline to ensure adequate time to travel to the root [23].

Fidelity eviction: The fidelity eviction policy delivers results based on a minimum window fidelity at the root. As panes arrive from nodes in the network, the root updates C^2 for the current window. When the fidelity reaches the bound the root merges the existing panes in

the window and outputs the answer.

Failure eviction: Just as the system evicts results that are 100% complete, the system may also evict results if additional wait time can not improve fidelity. This occurs when nodes are heavily loaded or become disconnected or fail. iMR employs *boundary* panes (where traditional stream processors use boundary tuples [26]) to distinguish between failed nodes and stalled or empty data streams[2]. Nodes periodically issue boundary panes to their parents when panes have been skipped because of a lack of data or load shedding (Section 3.4).

Boundary panes allow the root to distinguish between missing data that may arrive later and missing data that will never arrive. iMR maintains boundary information on a per-pane basis using two counters. The first counter is the C^2 completeness count; the number of successful pane merges. Even if a child has no local data for a pane, its parent in the aggregation tree may increase the completeness count for this pane. However, children may skip panes either because they re-started later in the stream (Section 4.4.2) or because they canceled processing to shed load (Section 3.4). In these cases, the parent node increases an *incompleteness* counter indicating the number of nodes that will never contribute to this pane.

Both interior nodes and the root use these counts to evict panes or entire windows respectively. Interior nodes evict early if the panes are complete or the sum of these two counters is equal to the sum of the children in this sub tree. The root determines whether or not the user's fidelity bound can ever be met. By simply subtracting incompleteness from the total node count (perfect completeness), the root can set an upper bound on C^2 for any particular window. If this estimate of C^2 ever falls below the user's target, the root evicts the window.

Note that the use of fidelity and latency bounds presumes that the user either received a usable result or cannot wait longer for it to improve. Thus, unlike other approaches, such as tentative tuples [8] or re-running the reduction phase [10], iMR does not, by default, update evicted results. iMR only supports this mode for debugging or determining a proper latency bound, as it can be expensive, forcing the system to repeatedly re-process (re-reduce) a window on late updates.

3.4 Load cancellation and shedding

When the root evicts incomplete windows, nodes in the aggregation tree may still be processing panes for that window. This may be due to panes with inordinate amounts of data or servers that are heavily loaded (have little time for log processing). Thus they are computing and merging panes that, once they arrive at the root, will no longer be used. This section discusses mechanisms that cancel or shed the work of creating and merging panes in the aggregation tree. Note that iMR assumes that mechanisms already exist to apportion server resources between the server's normal duties and iMR jobs. For instance, iMR may run in a separate virtual machine, letting the VM scheduler allocate resources between log processing and VMs running site services. Here our goal is to ensure that iMR nodes use the resources they are given effectively.

iMR's load cancellation policies try to ensure that internal nodes do not waste cycles creating or merging panes that will never be used. When the root evicts a window because it has met the minimum C^2 fidelity requirement, there is almost surely outstanding work in the network. Thus, once the root determines that it will no longer use a pane, it relays that pane's index down the aggregation tree. This informs the other nodes that they may safely stop processing (creating/merging) the pane.

In contrast, iMR's load shedding strategy works to prevent wasted effort when individual nodes are heavily loaded. Here nodes observe their local processing rates for creating a pane from local log records. If the expected time to completion exceeds the user's latency bound, it will cancel processing for that pane. It will then estimate the next processing deadline that it can meet and skip the intervening panes (and send boundary panes in their place).

Internal nodes also spend cycles (and memory) merging panes from children in the aggregation tree. Here interior nodes either choose to proceed with pane merging or, in the event that it violates the user's latency bound, "fast forward" the pane to its immediate parent. As we shall see in Section 5, these policies can improve result fidelity in the presence of straggler nodes.

4 Prototype

Our implementation of in-situ MapReduce builds upon Mortar, a distributed stream processing system [22]. We significantly extended Mortar's core functionality to support the semantics of iMR and the MapReduce programming model along four axes:

- Implement the iMR MapReduce API using generic map and reduce Mortar operators.
- Pane-based continuous processing with flow control.
- Load shedding/cancellation and pane/window eviction policies.
- Fault-tolerance mechanisms, including operator restart and adaptive tuple routing schemes.

[2] In reality, all panes contain boundary meta data, but nodes may issue panes that are otherwise empty except for this meta data.

4.1 Building an in-situ MapReduce query

Mortar computes continuous in-network aggregates across federated systems with thousands of nodes. This is a natural fit for the map, combine, and reduce functions since they are either local per-tuple transforms (map) or often in-network aggregates. A Mortar query consists of a single operator, or aggregate function, which Mortar replicates across nodes that produce the raw data streams. These in-situ operators give iMR the opportunity to actively filter and reduce intermediate data before it is sent across the network. Each query is defined by its operator type and produces a single, continuous output data stream. Operators push, as opposed to the pull-based method used in Hadoop, tuples across the network to other operators of the same type.

Mortar supports two query types: local and in-network queries. A local query processes data streams independently at each node. In contrast, in-network queries use a tree of operators to aggregate data across nodes. Either query type may subscribe to a local, raw data source such as a log file, or to the output of an existing query. Users compose these query types to accomplish more sophisticated tasks, such as MapReduce jobs.

Figure 6 illustrates an iMR job that consists of a local query for map operators and an in-network query for reduce operators. Map operators run on the log servers and partition their output among co-located reduce operators (here there are two partitions, hence two reduce trees). The reduce operator does most of the heavy lifting, grouping key-value pairs issued by the map operators before calling the user's combine, uncombine, and reduce functions. Unlike traditional MapReduce architectures, where the number of reducers is fixed during execution, iMR may dynamically add (or subtract) reducers during processing.

4.2 Map and reduce operators

Like other stream processors, Mortar uses processing windows to bound computation and provides a simple API to facilitate programming continuous operators. We implemented generic map and reduce operators using this API to call user-defined MapReduce functions at the appropriate time and properly group the key-value pairs. We modified operator internals so that they operate on panes as described in Section 2.3. Operators take as input either raw records from a local log or they receive panes from upstream operators in the aggregation tree. Internally, iMR represents panes as (possibly sorted) hash maps to facilitate key-value grouping.

In iMR operators have two main tasks: pane creation, creating an initial pane from a local data source, and pane merging, combining panes from children in an aggregation tree. Pane creation operates on a record-by-record basis, adding new records into the current pane. In contrast, pane merging combines locally produced panes with those arriving from the network. Because of differences in processing time and network congestion, operators maintain a sequence of panes that the system is actively merging (they have not yet been evicted). We call this the active pane list or APL.

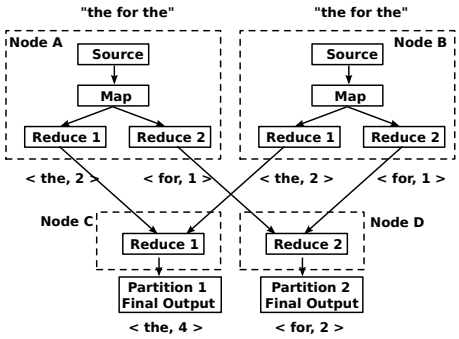


Figure 6: Each iMR job consists of a Mortar query for the map and a query for the reduce. Here there are two MapReduce partitions ($r = 2$), which result in two aggregation trees. A word count example illustrates partitioning map output across multiple reduce operators.

To adapt Mortar for MapReduce processing, we introduce immutable timestamps into the system. Mortar assumes logically independent operators that timestamp output tuples at the moment of creation. In contrast, iMR defines processing windows with respect to the original timestamps on the input logs, not with respect to the time at which an operator was able to evict a pane. iMR assigns a timestamp to each data record when it first enters the system (using a pre-existing timestamp from the log entry, or the current real time). This timestamp remains with the data as it travels through successive queries. Thus networking or processing delays do not alter the window in which the data belongs.

4.2.1 The map operator

The simplicity of mapping allows a streamlined map operator. The operator calls the user's map function for each arriving tuple, which may contain one or more log entries[3]. For each tuple, the map operator emits zero or more key-value pairs. We optimized the map operator by permanently assigning it a tuple window with a range and slide equal to one. This allowed us to remove window-related buffering and directly issue tuples containing key-value pairs to subscribed operators. Finally,

[3]Like Hadoop, iMR includes handlers that interpret log records.

the map operator partitions key-value pairs across subscribed reduce operators.

4.2.2 The reduce operator

The reduce operator handles the in-network functionality of iMR including the grouping, combining, sorting and reducing of key-value pairs. The operators maintain a hash map for each pane in the active pane list. Here we describe how the reduce operator creates and merges panes.

After a reduce operator subscribes to a local map operator it begins to receive tuples (containing key-value {k,v} pairs). The reducer operator first checks the logical timestamp of each {k,v} pair. If it belongs to the current pane, the system inserts the pair into the hash table and calls the combiner (if defined). When a {k,v} pair arrives with a timestamp for the next pane, the system inserts the prior pane into the active-pane list (APL). The operator may skip panes for which there is no local data. In that case, the operator inserts boundary panes into the APL with completeness counts of one.

Load shedding occurs during pane creation. As tuples arrive, the operator maintains an estimate of when the pane will complete. The operator periodically updates this estimate, maintained as an Exponentially Weighted Moving Average (EWMA) biased towards recent observations ($\alpha = 0.8$), and determines whether the user's latency deadline will be met. For accuracy, the operator processes 30% of the pane before the first estimate update. For responsiveness, the operator periodically updates and checks the estimate (every two seconds). For each skipped pane the operator issues a boundary pane with an incompleteness count of one.

The APL merges locally produced panes with panes from other reduce operators in the aggregation tree. The reduce operator calls the user's combiner for any group with new keys in the pane's hash map. The operator periodically inspects the APL to determine whether it should evict a pane (based on the policies in Section 3.3). Reduce operators on internal or leaf nodes forward the pane downstream on eviction.

If the operator is at the tree's root, it has the additional responsibility of determining when to evict the entire window. The operator checks eviction policies on periodic timeouts (the user's latency requirement) or when a new pane arrives (possibly meeting the fidelity bound). At that point, the operator may produce the final result either by using the optional uncombine function or by simply combining the constituent panes (strategies discussed in Section 2.3). After this combining step, the operator calls the user-defined reduce function for each key in the window's hash map.

4.3 Pane flow control

Recall that the goal of load shedding in iMR isn't to use less resources, but to use the given resources effectively. Given a large log file, load shedding changes the work done, not its processing rate. Thus it is still possible for some nodes to produce panes faster than others, either because they have less data per pane or more cycles available. In these cases, the local active pane list (APL) could grow in an unbounded fashion, consuming server memory and impacting its client-facing services.

We control the amount of memory used by the APL by employing a window-oriented flow control scheme. Each operator monitors the memory used (by the JVM in our implementation) and issues a pause indicator when it reaches a user-defined limit. The indicator contains the logical index of the youngest pane in the operator's APL. Internally, pane creation waits until the indicator is greater than the current index or the indicator is removed. Pause indicators are also propagated top-down in the aggregation tree, ensuring that operators send evicted panes upward only when the indicator is greater than the evicted indices or it is not present.

4.4 MapReduce with gap recovery

While load shedding and pane eviction policies improve availability during processing and network delays, nodes may fail completely, losing their data and current queries. While traditional MapReduce designs, such as Hadoop, can restart map or reduce tasks on any node in the cluster, iMR does not assume a shared filesystem. Instead, iMR provides *gap recovery* [19], meaning that the system may drop tuples (i.e., panes) in the event of node failures.

4.4.1 Multi-tree aggregation

Mortar avoids failed network elements and nodes by routing data up multiple trees. Nodes route data up a single tree until the node stops receiving heart beats from its parent. If a parent becomes unreachable, it chooses another tree (i.e., another parent) to route tuples to. For this work, we use a single tree; this simplifies our implementation of failure eviction policies because internal nodes know the maximum possible completeness of panes arriving from their children.

Mortar employs new tuple routing rules to retain a degree of failure resilience. If a parent becomes unreachable, the child forwards data directly to the root. This policy allows data to bypass failed nodes at the expense of fewer aggregation opportunities. Mortar also designs its trees by clustering network coordinates [11], and we use the same mechanism in our experiments. We leave more advanced routing and tree-building schemes as future work.

4.4.2 Operator re-install

iMR guarantees that queries (operators) will be installed and removed on nodes in an eventually consistent manner. Mortar provides a reconciliation algorithm to ensure that nodes eventually install (or un-install) query operators. Thus, when nodes recover from a failure, they will re-install their current set of operators. While we lose the data in the operator's APL at the time of failure, we need to re-start processing at an appropriate point to avoid duplicate data. To do so, operators, during pane creation, maintain a simple on-disk write-ahead log to indicate the next safe point in the log to begin processing on re-start. For many queries the cost of writing to this log is small relative to pane computation, and we simply point to the next pane.

5 Evaluation

Our evaluation explores both the baseline performance of our prototype and the ability of our system to deliver results in the event of delays or failures. Unless noted otherwise, we evaluated iMR on a 40 node cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24 GB of memory, and 16 HP 507750-B21 500GB 7,200 RPM 2.5" SATA drives. Each server has two HP P410 drive controllers, as well as a Myricom 10 Gbps network interface. The network interconnect we use is a 52-port Cisco Nexus 5020 datacenter switch. The servers run Linux 2.6.35, and our implementation of iMR is written in Java. iMR experiments use star aggregation topologies.

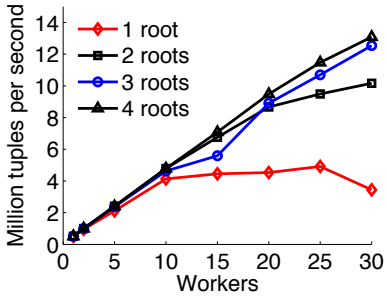
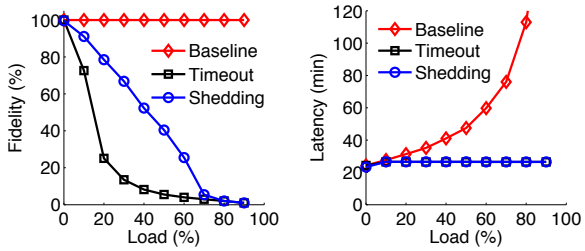


Figure 7: Scaling iMR as the number of workers (and processing nodes) increases.

5.1 Scaling

We first establish the scale-out properties of our processing architecture. The purpose of these experiments is to verify the ability of the system to scale as we increase both the number of mappers and the number of reducer partitions. Here we use synthetic input data and a reducer that implements a word count function. The query uses a tumbling window where the range is equal to the slide; in this case the window range is 150 million input records, approximately 1GB of input data. We allow the job to run for five minutes and take the average throughput. Unlike Hadoop, the iMR job is configured to read the log from local disk.

Figure 7 plots the records per second throughput of iMR as we increase the total cluster capacity. Each line represents a different configuration that increases the reducer and physical node count by one. Here three reducers provide sufficient processing to handle the 30 map tasks. We see that, as long as the reducer is not the bottleneck, adding additional nodes increases throughput linearly. Similarly, reducers can also add a linear increase in throughput.



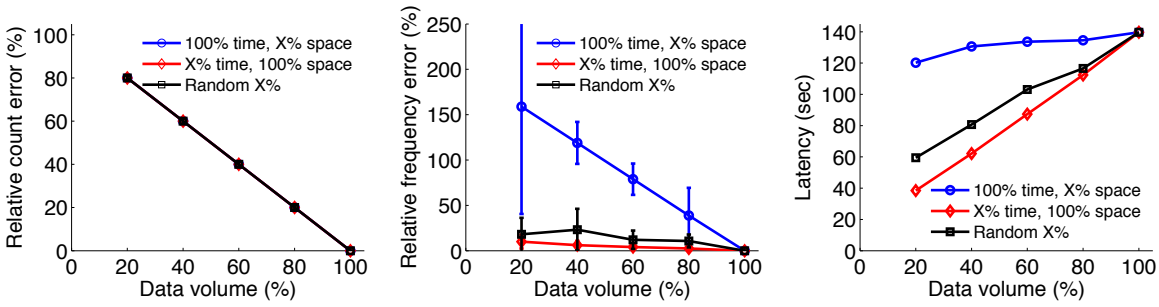
(a) Fidelity (b) Latency

Figure 8: Impact of load shedding on fidelity and latency for a word count job under maximum latency requirement and varying worker load.

5.2 Load shedding

These iMR experiments evaluate the ability of load shedding to improve result fidelity under limited CPU resources. We execute a word count MapReduce query on a single node; this node installs a single map and reduce operator. We vary the CPU load by running a separate CPU burn application. The query specifies a tumbling window ($R = S$) that contains 20 million records and we configure the system to use 20 panes per window. We execute the query until it delivers 10 results and report the average latency (Figure 8(a)) and fidelity (Figure 8(b)) as we increase CPU load.

The baseline query has no latency requirement and always delivers results with 100% fidelity. The timeout query has a latency requirement equal to the observed baseline window latency, which is 160 seconds. Though results meet the latency requirement, quality degrades as the load increases. Without load shedding the worker attempts to process all panes, even if very few can be delivered in time. In contrast, load shedding allows the



(a) Count error (b) Frequency error. (c) Result latency.

Figure 9: The performance of a count statistic on data skewed across the log server pool. Enforcing either random pane selection or spatial completeness allows the system to approximate count frequencies and lower result latency.

worker to use the available CPU intelligently, processing only the panes that can be delivered on time and increasing average fidelity substantially.

5.3 Failure eviction

Here we show how failure eviction can deliver results early if nodes fail. We execute a word count MapReduce query on 10 workers. The query uses a tumbling window with 2 million records, 2 panes, and a 30 second latency requirement. After starting the query, we emulate transient failures by stopping an increasing number of workers. The experiment finishes when the query delivers 20 results.

In Figure 10, we report application goodput as the number of panes delivered to the user per time. Note that this metric is not a direct measure of how fast workers can process raw data. Instead it reflects the ability of the system to detect failures and deliver panes to the user early. The higher the metric, the less the user waits to get the same number of panes. Without failure eviction the root times out (30 seconds) before it delivers incomplete results. With failure eviction, the root can deliver results before the timeout, improving goodput by 57-64%.

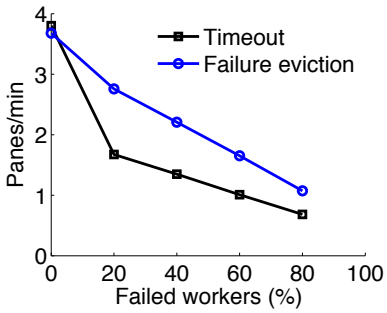


Figure 10: Application goodput as the percentage of failed workers increases. Failure eviction delivers panes earlier, improving goodput by up to 64%.

5.4 Using C^2

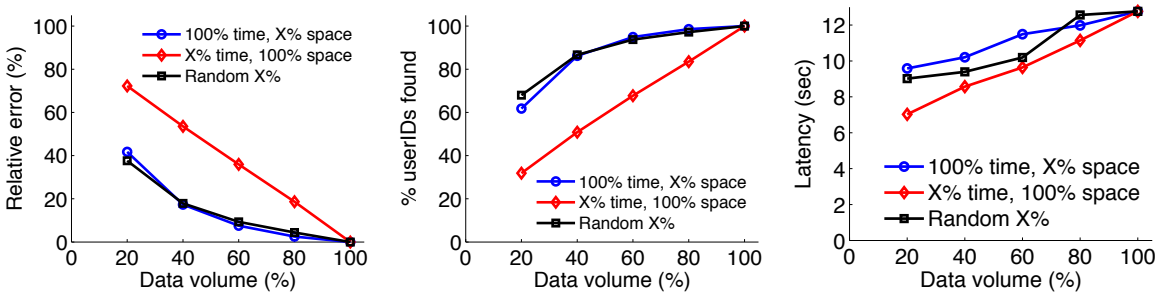
This section explores how we use the C^2 framework for three different application scenarios: word count with non-uniformly distributed keys, click-stream analysis, and an HDFS anomaly detector. These experiments used a 30-node cluster of Dual Intel Xeon 2.4GHz machines with 4GB of RAM connected by gigabit Ethernet.

5.4.1 Word Count

Our first experiment performed a word count query across synthetic data placed on ten log servers in our local cluster. This configuration allows us to explore the impact of different fidelity bounds on absolute count estimations and relative word frequency. We distribute the words in the synthetic data across the log servers in a skewed fashion, where some words are more likely to be on some servers than others. In these experiments the window range (and slide) is 100MB, the pane size is 10 MB, and there is no latency bound.

Here we explore three different C^2 settings: temporal completeness, spatial completeness, and area with random pane selection. Figure 9 shows the relative error in reported count, the relative error of the word frequency (with std. dev.), and the result latency as we increase the data fidelity. As expected, the count error (Figure 9(a)) improves linearly as we force the system to include more data in each window (data volume).

However, because the data are not uniformly distributed, the frequency error (Figure 9(b)) is large for the temporal completeness C^2 specification, $(100\%, Y)$. Note in this experiment we achieve varying levels of temporal completeness by randomly selecting specific nodes to fail to report for an entire window. By removing data from a source completely, some keys may completely lose their representation and the remaining key's fre-



(a) Average error per userID (b) Percentage userID's found. (c) Result latency.

Figure 11: Estimating user session count using iMR and different C^2 policies. Random pane selection and temporal completeness provide significantly higher data fidelity than enforcing spatial completeness.

quencies shift. Both random pane selection and spatially complete results do much better, since they effectively sample from the entire server pool.

Finally, these three policies differ substantially in the latency of the results they deliver. Figure 9(c) plots the result latency for each C^2 specification. Clearly, providing temporal completeness requires each node to finish processing the entire window before returning a result. In contrast, by asking for spatial completeness, the root can return as soon as the first x% of the panes complete, allowing the best latency.

5.4.2 Click-stream analysis

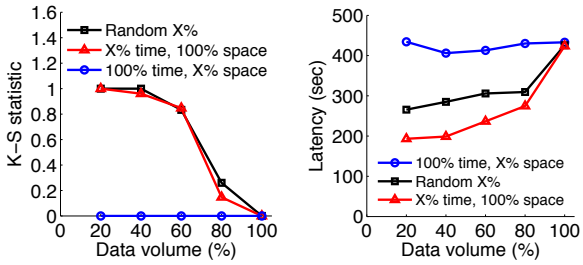
Here we develop a simple click-stream analysis. This analysis takes as input a log of click records that contain userID and timestamp fields. We developed a reduce function to calculate three different click analysis metrics: the number of user sessions, the average session duration, and the average number of clicks per session. We use our different C^2 specifications and study the relative error each provides.

These experiments use 24 hours of publicly available server logs from the 1998 World Cup [1] as input. We partition this data (4.5GB in total) across ten of our servers, preserving the characteristic that clicks from a single user are often served by different nodes in the trace. The window (and slide) of the MapReduce job is set to two hours and we set the pane size to be 6 minutes (20 panes per window). We run each query for the entire data set (12 windows).

Figure 11 shows how the number of sessions per user changes as we accept different levels of data fidelity. Surprisingly, requiring data from all nodes for each pane, $(X, 100\%)$, leads to large relative errors (per user). This is primarily because userIDs are not uniformly distributed across time and enforcing spatial completeness does not give a decent sample. However, randomly sampling at each log server lowers relative error to 20% (per user), even when computing across less than 50% of the window's data.

Figure 11(b) shows that those policies also recover a large fraction of the total userID space even when they sample a relatively small total fraction of data. Thus for this application, the best C^2 specification is random pane selection, as it not only provides the best results but also allows the system to lower result latency as well (Figure 11(c)).

5.4.3 HDFS log analysis



(a) KS-test (b) Latency

Figure 12: (a) Results from the Kolmogorov-Smirnov test illustrate the impact of reduced data fidelity on the histograms reported for each HDFS server. (b) For HDFS anomaly detection, random and spatial completeness C^2 improve latency by at least 30%.

Our last application analyzes logs from the Hadoop distributed file system (HDFS) to determine faulty storage nodes. The iMR MapReduce job first filters the local HDFS log, finding all unique block write events. The reduce function then computes a histogram of the block write service times. This collection of histograms, one per HDFS server, is then analyzed to determine anomalies in the cluster [27].

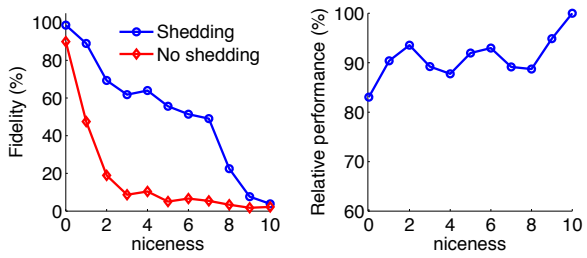
We generated 48 hours of HDFS logs by running the

GridMix Hadoop workload generator [3] on our 30-node cluster. Each node's log is approximately 2.5 GB, yielding appr. 75 GB in total. This analysis compares the quality of the histograms produced under different C^2 specifications to the histogram produced with no loss. The query has a window range (and slide) of 48 hours and uses 1 hour panes.

We use the Kolmogorov-Smirnov test to compare the per-server histograms with perfect and incomplete data. Figure 12(a) shows the percentage of histograms that when using incomplete data represent a markedly different distribution (reject the null hypothesis). Here the $(100\%, Y)$ policy generates perfect data, since, if a node reports, all data is included. The other C^2 strategies result in a majority of the histograms failing the null hypothesis when using less than 80% of the data.

However, since those strategies can lower result latency significantly at that data volume (about 30% in Figure 12(b)), users must decide whether that is an acceptable tradeoff. Going forward we intend to look at how this ultimately impacts the ability to find failing HDFS nodes.

5.5 In-situ performance



(a) Fidelity (b) Hadoop performance

Figure 13: Fidelity and Hadoop performance as a function of the iMR process niceness. Hadoop is always given the highest priority, nice $= 0$.

We designed iMR to effectively process log data "on location." This experiment illustrates the ability of the iMR architecture to produce useful results when run side-by-side with a real application. Specifically, our 10-node cluster will execute Hadoop and iMR simultaneously. Here, Hadoop executes a workload generated by the GridMix generator and iMR executes a word count query with a window of 2 million records, 20 panes per window, and a 60 second timeout. We vary the CPU allocated to iMR by changing the priority (niceness) assigned to the iMR process by the kernel scheduler and report the average result fidelity. We also report the relative change in the Hadoop performance, in terms of jobs completed per time. Each data point is the average of five runs.

Figure 13(a) shows that without load shedding, result fidelity falls almost linearly as the iMR process' priority decreases. In contrast, load shedding greatly improves fidelity until there is insufficient CPU remaining to process any pane by the deadline (nice $= 9$). Looking at Hadoop performance in Figure 13(b), we see that the cost for giving them equal priorities is a decrease in job throughput of 17%. Even when using nice, a relatively coarse-grain knob for resource allocation, to assign a lower priority to log processing, Hadoop can improve job throughput ($< 10\%$ penalty) and iMR can still deliver useful results.

6 Related work

"Online" bulk processing: iMR focuses on the challenges of migrating initial log analytics to the data sources. A different (and complementary) approach has been to optimize traditional MapReduce architectures for log processing themselves. For instance, the Hadoop Online Prototype (HOP) [10] can run continuously, but requires custom reduce functions to manage their own state for incremental computation and framing incoming data into meaningful units (windows). iMR's design avoids this requirement by explicitly supporting sliding window-based computation (Section 2.1), allowing existing reduce functions to run continuously without modification.

Like iMR, HOP also allows incomplete results, producing "snapshots" of reduce output, where the reduce phase executes on the map output that has accumulated thus far. HOP describes incomplete results with a "progress" metric that (self admittedly) is often too coarse to be useful. In contrast, iMR's C^2 framework (Section 3) not only provides both spatial and temporal information about the result, but may be used to trade particular aspects of data fidelity for decreased processing time.

Dremel [24] is another system that, like iMR, aims to provide fast analysis on large-scale data. While iMR targets continuous raw log data, Dremel focuses on static nested data, like web documents. It employs an efficient columnar storage format that is benefitial when a fraction of the fields of the nested data must be accessed. Like HOP, Dremel uses a coarse progress metric for describing early, partial results.

Log collection systems: A system closely related to iMR is Flume [2], a distributed log collection system that places *agents* in-situ on servers to relay log data to a tier of collectors. While a user's "flows" (i.e., queries) may transform or filter individual events, iMR provides a more powerful data processing model with grouping, reduction, and windowing. While Flume supports best-

effort operation, users remain in the dark about result quality or latency. However, Flume does provide higher reliability modes, recovering events from a write-ahead log to prevent data loss. While not discussed here, iMR could employ similar *upstream backup* [19] techniques to better support queries that specify fidelity bounds.

Load shedding in data stream processors: iMR's load shedding (Section 3.4) and result eviction policies (Section 3.3) build upon the various load shedding techniques explored in stream processing [9, 28, 29]. For instance, iMR's latency and fidelity bounds are related to the QoS metrics found in the Aurora stream processor [9]. Aurora allows users to provide "graphs" which separately map increased delay and percent tuples lost with decreasing output quality (QoS). iMR takes a different approach, allowing users to specify latency and fidelity bounds above which they'd be satisfied. Additionally, iMR leverages the temporal and spatial nature of log data to provide users more control than percent tuples lost.

Many of these load shedding mechanisms insert tuple dropping operators into query plans and coordinate drop probabilities, typically via a centralized controller, to maintain result quality under high-load conditions. In contrast, our load shedding policies act locally at each operator, shedding sub-windows (panes) as they are created or merged. These "pane drop" policies are more closely related to the probabilistic "window drop" operators proposed by Tatbul, et al. [29] for aggregate operators. In contrast, iMR's operators may drop panes both deterministically or probabilistically depending on the C^2 fidelity bound.

Distributed aggregation: Aggregation trees have been explored in sensor networks [23], monitoring wired networks [31], and distributed data stream processing [18, 22]. More recent work explored a variety of strategies for distributed GroupBy aggregation required in MapReduce-style processing [32]. Our use of sub-windows (panes) is most closely related to their *Accumulator-PartialHash* strategy, since we accumulate (through combining) key-value pairs into each sub-window. While they evicted the sub window based on its storage size (experiencing a hash collision), iMR uses fixed-sized panes.

7 Conclusion

This work explores moving initial log analysis steps out of dedicated clusters and onto the data sources themselves. By leveraging continuous in-situ processing, iMR can efficiently extract and transform data, improving system scalability and reducing analysis times. A key challenge is to provide a characterization of result fidelity that allows users to interpret results in the face of incomplete data. For a handful of applications, we illustrated how the C^2 framework allows users to explicitly trade specific aspects of data fidelity in the event failures lose data or the system cannot meet latency requirements. Future work will consider how the system can assist in setting appropriate C^2 fidelity bounds, and whether similar techniques could be applied in dedicated processing cluster environments.

Acknowledgements

We'd like to thank Geoff Voelker and our shepherd, Leendert van Doorn, for their helpful feedback. This work was supported in part by the National Science Foundation through grant CCF-1048296.

References

[1] 1998 World Cup Web Server Logs. http://ita.ee.lbl.gov/html/traces.html.

[2] Flume: Open source log collection system. http://github.com/cloudera/flume.

[3] The GridMix Hadoop Workload Generator. http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[4] Windows Azure and Facebook teams. Personal communications, August 2008.

[5] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 2005.

[6] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *NSDI'10*, April 2010.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Symposium on Principles of Database Systems*, March 2002.

[8] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the Borealis distributed stream processing system. In *SIGMOD'05*, June 2005.

[9] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams—a new class of data management applications. In *VLDB'02*, September 2002.

[10] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *NSDI'10*, San Jose, CA, April 2010.

[11] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM'04*, August 2004.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, San Francisco, CA, December 2004.

[13] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, alex Russakovsky, and N. Thomre. Continuous analytics: Rethinking query processing in a network-effect world. In *Proc. of CIDR*, January 2009.

[14] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1), 1997.

[15] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[16] B. He, M. Yang, zhenyu Guo, R. Chen, W. Lin, B. Su, and L. Zhou. Batched stream processing: A case for data intensive distributed computing. In *ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[17] T. Hoff. How Rackspace Now Uses MapReduce and Hadoop To Query Terabytes of Data, Jan 2008. http://highscalability.com/how-rackspace-now-uses-mapreduce-and-hadoop-query-terabytes-data.

[18] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of VLDB*, September 2003.

[19] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. of ICDE*, April 2005.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys'07*, March 2007.

[21] J. Li, D. Maier, K. Tufte, V. Papdimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1), March 2005.

[22] D. Logothetis and K. Yocum. Wide-scale data stream management. In *USENIX Annual Technical Conf.*, June 2008.

[23] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI'02*, December 2002.

[24] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel : Interactive Analysis of Web-Scale Datasets. *Proceedings of the VLDB Endowment*, 3, September 2010.

[25] R. N. Murty and M. Welsh. Towards a dependable architecture for Internet-scale sensing. In *Second HotDep06*, November 2006.

[26] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of PODS 2004*, June 2004.

[27] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: analyzing logs as state machines. In *1st USENIX Workshop on Analysis of System Logs*, page 6, December 2008.

[28] N. Tatbul, U. Cetintemel, and S. Zdonik. Staying FIT: Efficient load shedding techniques for distributed stream processing. In *VLDB'07*, August 2007.

[29] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB'06*, August 2006.

[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Mining console logs for large-scale system problem detection. In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques)*, Dec 2008.

[31] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *SIGCOMM'04*, September 2004.

[32] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP'09*, October 2009.

Exception-Less System Calls for Event-Driven Servers

Livio Soares Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto

Abstract

Event-driven architectures are currently a popular design choice for scalable, high-performance server applications. For this reason, operating systems have invested in efficiently supporting non-blocking and asynchronous I/O, as well as scalable event-based notification systems.

We propose the use of *exception-less system calls* as the main operating system mechanism to construct high-performance event-driven server applications. Exception-less system calls have four main advantages over traditional operating system support for event-driven programs: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor based, (2) support in the operating system kernel is non-intrusive as code changes are not required for each system call, (3) processor efficiency is increased since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multi-core execution for event-driven programs is easier, given that a single user-mode execution context can generate enough requests to keep multiple processors/cores busy with kernel execution.

We present *libflexsc*, an asynchronous system call and notification library suitable for building event-driven applications. Libflexsc makes use of exception-less system calls through our Linux kernel implementation, FlexSC. We describe the port of two popular event-driven servers, memcached and nginx, to libflexsc. We show that exception-less system calls increase the throughput of memcached by up to 35% and nginx by up to 120% as a result of improved processor efficiency.

1 Introduction

In a previous publication, we introduced the concept of **exception-less system calls** [28]. With exception-less system calls, instead of issuing system calls in the traditional way using a trap (exception) to switch to the kernel for the processing of the system call, applications issue kernel requests by writing to a reserved **syscall page**, shared between the application and the kernel, and then switching to another user-level thread ready to execute without having to enter the kernel. On the kernel side, special in-kernel **syscall threads** asynchronously execute the posted system calls obtained from the shared syscall page, storing the results to the syscall page after their servicing. This approach enables flexibility in the scheduling of operating system work both in the form of kernel request batching, and (in the case of multi-core processors) in the form of allowing operating system and application execution to occur on different cores. This not only significantly reduces the number of costly domain switches, but also significantly increases temporal and spacial locality at both user and kernel level, thus reducing pollution effects on processor structures.

Our implementation of exception-less system calls in the Linux kernel (**FlexSC**) was accompanied by a user-mode POSIX compatible thread package, called **FlexSC-Threads**, that transparently translates legacy synchronous system calls into exception-less ones. FlexSC-Threads primarily targets highly threaded server applications, such as Apache and MySQL. Experiments demonstrated that FlexSC-Threads increased the throughput of Apache by 116% while reducing request latencies by 50%, and increased the throughput of MySQL by 40% while reducing request latencies by roughly 30%, requiring no changes to these applications.

In this paper we report on our subsequent investigations on whether exception-less system calls are suitable for event-driven application servers and, if so, whether exception-less system calls are effective in improving throughput and reducing latencies. Event-driven application server architectures handle concurrent requests by using just a single thread (or one thread per core) so as to reduce application-level context switching and the memory footprint that many threads otherwise require. They make use of non-blocking or asynchronous system calls to support the concurrent handling of requests. The belief that event-driven architectures have superior perfor-

mance characteristics is why this architecture has been widely adopted for developing high-performant and scalable servers [12, 22, 23, 26, 30]. Widely used application servers with event-driven architectures include memcached and nginx.

The design and implementation of operating system support for asynchronous operations, along with event-based notification interfaces to support event-driven architectures, has been an active area of both research and development [4, 8, 7, 11, 13, 14, 16, 22, 23, 30]. Most of the proposals have a few common characteristics. First, the interfaces exposed to user-mode are based on file descriptors (with the exception of *kqueue* [4, 16] and LAIO [11]). Consequently, resources that are not encapsulated as descriptors (e.g., memory) are not supported. Second, their implementation typically involved significant restructure of kernel code paths into an asynchronous state-machine in order to avoid blocking the user execution context. Third, and most relevant to our work, while the system calls used to request operating system services are designed not to block execution, applications still issue system calls *synchronously*, raising a processor exception, and switching execution domains, for every request, status check, or notification of completion.

In this paper, we demonstrate that the exception-less system call mechanism is well suited for the construction of event-based servers and that the exception-less mechanism presents several advantages over previous event-based systems:

1. **General purpose.** Exception-less system call is a general mechanism that supports any system call and is not necessarily tied to operations with file descriptors. For this reason, exception-less system calls provide asynchronous operation on any operating system managed resource.
2. **Non-intrusive kernel implementation.** Exception-less system calls are implemented using light-weight kernel threads that can block without affecting user-mode execution. For this reason, kernel code paths do not need to be restructured as asynchronous state-machines; in fact, no changes are necessary to the code of standard system calls.
3. **Efficient user and kernel mode execution.** One of the most significant advantages of exception-less system calls is its ability to decouple system call invocation from execution. Invocation of system calls can be done entirely in user-mode, allowing for truly asynchronous execution of user code. As we show in this paper, this enables significant performance improvements over the most efficient non-blocking interface on Linux.
4. **Simpler multi-processing.** With traditional system calls, the only mechanism available for applications to exploit multiple processors (cores) is to use an operating system visible execution context, be it a thread or a process. With exception-less system calls, however, operating system work can be issued and distributed to multiple remote cores. As an example, in our implementation of memcached, a single memcached thread was sufficient to generate work to fully utilize 4 cores.

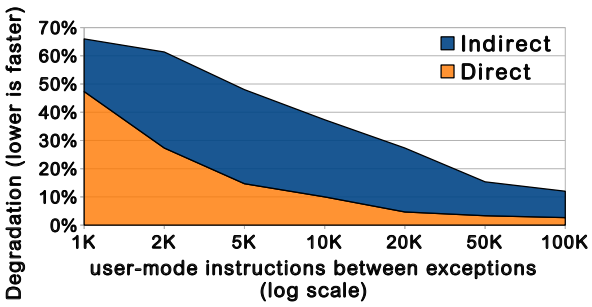


Figure 1: System call (`pwrite`) impact on user-mode instructions per cycle (IPC) as a function of system call frequency for Xalan.

Specifically, we describe the design and implementation of an asynchronous system call notification library, **libflexsc**, which is intended to efficiently support event-driven programs. To demonstrate the performance advantages of exception-less system calls for event-driven servers, we have ported two popular and widely deployed event-based servers to libflexsc: memcached and nginx. We briefly describe the effort in porting these applications to libflexsc. We show how the use of libflexsc can significantly improve the performance of these two servers over their original implementation using non-blocking I/O and Linux's `epoll` interface. Our experiments demonstrate throughput improvements in memcached of up to 35% and nginx of up to 120%. As anticipated, we show that the performance improvements largely stem from increased efficiency in the use of the underlying processor.

2 Background and Motivation: Operating System Support for I/O Concurrency

Server applications that are required to efficiently handle multiple concurrent requests rely on operating system primitives that provide I/O concurrency. These primitives typically influence the programming model used to implement the server. The two most commonly used models for I/O concurrency are threads and non-blocking/asynchronous I/O.

Thread based programming is often considered the simplest, as it does not require tracking the progress of I/O operations (which is done implicitly by the operating system kernel). A disadvantage of threaded servers that utilize a separate thread per request/transaction is the inefficiency

Server (workload)	Syscalls per Request	User Instructions per Syscall	User IPC	Kernel Instructions per Syscall	Kernel IPC
Memcached (memslap)	2	3750	0.80	5420	0.59
nginx (ApacheBench)	12	1460	0.46	6540	0.49

Table 1: The average number of instructions executed on different workloads before issuing a syscall, the average number of system calls required to satisfy a single request, and the resulting processor efficiency, shown as instructions per cycle (IPC) of both user and kernel execution. Memcached and nginx are event-driven servers using Linux's `epoll` interface.

of handling a large number of concurrent requests. The two main sources of inefficiency are the extra memory usage allocated to thread stacks and the overhead of tracking and scheduling a large number of execution contexts.

To avoid the overheads of threading, developers have adopted the use event-driven programming. In an event-driven architecture, the program is structured as a state machine that is driven by progress of certain operations, typically involving I/O. Event-driven programs make use of non-blocking or asynchronous primitives, along with event notification systems, to deal with concurrent I/O operations. These primitives allow for uninterrupted execution that enables a single execution context (e.g., thread) to fully utilize the processor. The main disadvantage of using non-blocking or asynchronous I/O is that it entails a more complex programming model. The application is responsible for tracking the status of I/O operations and availability of I/O resources. In addition, the application must support multiplexing the execution of stages of multiple concurrent requests.

In both models of I/O concurrency, the operating system kernel plays a central role in supporting servers in multiplexing execution of concurrent requests. Consequently, to achieve efficient server execution, it is critical for the operating system to expose and support efficient I/O multiplexing primitives. To quantify the relevance of operating system kernel execution experimentally, we measured key execution metrics of two popular event-driven servers: memcached and nginx.

Table 1 shows the number of instructions executed in user and kernel mode, on average, before changing mode, for nginx and memcached. (Sections 5 and 6 explain the servers and workloads in more detail.) These applications use non-blocking I/O, along with the Linux `epoll` facility for event notification. Despite the fact that the `epoll` facility is considered the most scalable approach to I/O concurrency on Linux, management of both I/O requests and events is inherently split between the operating system kernel and the application. This fundamental property of event notification systems imply that there is a need for continuous communication between the application and the operating system kernel. In the case of nginx, for example, we observe that communication with the kernel occurs, on average, every 1470 instructions.

We argue that the high frequency of mode switching in these servers, which is inherent to current event-based facilities, is largely responsible for the low efficiency of user and kernel execution, as quantified by the instructions per cycle (IPC) metric in Table 1. In particular, there are two costs that affect the efficiency of execution when frequently switching modes: (1) a *direct* cost that stems from the processor exception associated with the system call instruction, and (2) an *indirect* cost resulting from the pollution of important processor structures.

To quantify the performance interference caused by frequent mode switching, we used the Core i7 hardware performance counters (HPC) to measure the efficiency of processor execution while varying the number of mode switches of a benchmark. Figure 1 depicts the performance degradation of user-mode execution, when issuing varying frequencies of `pwrite` system calls, on a high IPC workload, Xalan, from the SPEC CPU 2006 benchmark suite. We used a benchmark from SPEC CPU 2006 as these benchmarks have been created to avoid significant use of system services, and should spend only 1-2% of time executing in kernel-mode. Xalan has a baseline user-mode IPC of 1.46, but the IPC degrades by up to 65% when executing a `pwrite` every 1,000-2,000 instructions, yielding an IPC between 0.50 and 0.58.

The figure also depicts the breakdown of user-mode IPC degradation due to direct and indirect costs. The degradation due to the direct cost was measured by issuing a null system call, while the indirect portion is calculated by subtracting the direct cost from the degradation measured when issuing a `pwrite` system call. For high frequency system call invocation (once every 2,000 instructions, or less, which is the case for nginx), the direct cost of raising an exception and subsequent flushing of the processor pipeline is the largest source of user-mode IPC degradation. However, for medium frequencies of system call invocation (once per 2,000 to 100,000 instructions), the *indirect* cost of system calls is the dominant source of user-mode IPC degradation.

Given the inefficiency of event-driver server execution due to frequent mode switches, we envision that these servers could be adapted to make use of exception-less system calls. Beyond the potential to improve server performance, we believe exception-less system calls is an appealing mechanism for event-driven programming, as: (1) it is as simple as asynchronous I/O to program to (no retry logic is necessary, unlike non-blocking I/O), and (2) more generic than asynchronous I/O, which mostly

supports descriptor based operations and which are only partially supported on some operating systems due to their implementation complexity (e.g., Linux does not offer an asynchronous version of the zero-copy `sendfile()`).

One of the proposals that is closest to achieving the goals of event-driven programming with exception-less system calls is *lazy asynchronous I/O* (LAIO), proposed by Elmeleegy et al. [11] However, in their proposal, system calls are still issued synchronously, using traditional exception based calls. Furthermore, a completion notification is also needed whenever an operation blocks, which generates another interruption in user execution.

3 Exception-Less System Call Interface and Implementation

In this work, we argue for the use of exception-less system calls as a mechanism to improve processor efficiency while multiplexing execution between user and kernel modes in event-driven servers. Exception-less system call is a mechanism for requesting kernel services that does not require the use of synchronous processor exceptions [28]. The key benefit of exception-less system calls is the flexibility in scheduling system call execution, ultimately providing improved locality of execution of both user and kernel code.

Exception-less system calls have been shown to improve the performance of highly threaded applications, by using a specialized user-level threading package that transparently converts synchronous system calls into exception-less ones [28]. The goal of this work is to extend the original proposal by enabling the explicit use of exception-less system calls by event-driven applications.

In this section, we briefly describe the original exception-less system call implementation (FlexSC) for the benefit of the reader; those readers already familiar with exception-less system calls may skip to Section 4. For space considerations, this is a simple overview of exception-less system calls; for more information, we refer the reader to the original exception-less system calls proposal [28].

3.1 Exception-Less System Calls

The design of exception-less system calls consists of two components: (1) an exception-less interface for user-space threads to register system calls, along with (2) an in-kernel threading system that allows the delayed (asynchronous) execution of system calls, without interrupting or blocking the thread in user-space.

3.1.1 Exception-Less Syscall Interface

The interface for exception-less system calls is simply a set of memory pages that is shared between user and kernel space. The shared memory pages, henceforth referred to as **syscall pages**, contain exception-less system call entries. Each entry records the request status, system call number, arguments, and return value (Figure 2).

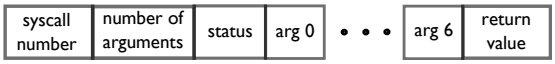
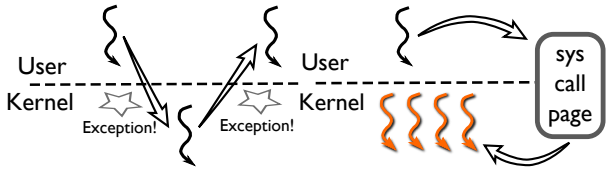


Figure 2: 64-byte syscall entry from the syscall page.



(a) Traditional, sync. system call (b) Exception-less system call

Figure 3: Illustration of synchronous and exception-less system call invocation. The left diagram shows the sequential nature of exception-based system calls, while the right diagram depicts exception-less user and kernel communication through shared memory.

The traditional invocation of a system call occurs by populating predefined registers with call information and issuing a specific machine instruction that immediately raises an exception. In contrast, to issue an exception-less system call, user-mode must find a free entry in the syscall page and populate the entry with the appropriate values using regular store instructions. The user thread can then continue executing without interruption. It is the responsibility of the user thread to later verify the completion of the system call by reading the status information in the entry. None of these operations, issuing a system call or verifying its completion, causes exceptions to be raised.

3.1.2 Decoupling Execution from Invocation

Along with the exception-less interface, the operating system kernel must support delayed execution of system calls. Unlike exception-based system calls, the exception-less system call interface does not result in an explicit kernel notification, nor does it provide an execution stack. To support decoupled system call execution, we use a special type of kernel thread, which we call **syscall thread**. Syscall threads always execute in kernel mode, and their sole purpose is to pull requests from syscall pages and execute them on behalf of the user thread. Figure 3 illustrates the difference between traditional synchronous system calls, and our proposed split system call model.

3.2 Implementation – FlexSC

Our implementation of the exception-less system call mechanism is called **FlexSC** (Flexible System Call) and was prototyped as an extension to the Linux kernel. Two

new system calls were added to Linux as part of FlexSC, `flexsc_register` and `flexsc_wait`.

`flexsc_register()` This system call is used by processes that wish to use the FlexSC facility. Registration involves two steps: mapping one or more syscall pages into user-space virtual memory space, and spawning one syscall thread per entry in the syscall pages.

`flexsc_wait()` The decoupled execution model of exception-less system calls creates a challenge in user-mode execution, namely what to do when the user-space thread has nothing more to execute and is waiting on pending system calls. The solution we adopted is to require that the user explicitly communicate to the kernel that it cannot progress until one of the issued system calls completes by invoking the `flexsc_wait` system call (this is akin to `aio_suspend()` or `epoll_wait()` calls). FlexSC will later wake up the user-space thread when at least one of the posted system calls are complete.

3.2.1 Syscall Threads

Syscall thread is the mechanism used by FlexSC to allow for exception-less execution of system calls. The Linux system call execution model has influenced some implementation aspects of syscall threads in FlexSC: (1) the virtual address space in which system call execution occurs is the address space of the corresponding process, and (2) the current thread context can be used to block execution should a necessary resource not be available (for example, waiting for I/O).

To resolve the virtual address space requirement, syscall threads are created during `flexsc_register`. Syscall threads are thus "cloned" from the registering process, resulting in threads that share the original virtual address space. This allows the transfer of data from/to user-space with no modification to Linux's code.

FlexSC would ideally never allow a syscall thread to sleep. If a resource is not currently available, notification of the resource becoming available should be arranged, and execution of the next pending system call should begin. However, implementing this behavior in Linux would require significant changes and a departure from the basic Linux architecture. Instead, we adopted a strategy that allows FlexSC to maintain the Linux thread blocking architecture, as well as requiring only minor modifications (3 lines of code) to the Linux context switching code, by creating multiple syscall threads for each process that registers with FlexSC.

In fact, FlexSC spawns as many syscall threads as there are entries available in the syscall pages mapped in the process. This provisions for the worst case where every pending system call blocks during execution. Spawning hundreds of syscall threads may seem expensive, but Linux in-kernel threads are typically much lighter weight than user threads: all that is needed is a `task_struct` and a small, 2-page, stack for execution. All the other structures (page table, file table, etc.) are shared with the user process. In total, only 10KB of memory is needed per syscall thread.

Despite spawning multiple threads, only *one* syscall thread is active per application and core at any given point in time. If a system call does not block, then all the work is executed by a single syscall thread, while the remaining ones sleep on a work-queue. When a syscall thread needs to block, for whatever reason, immediately before it is put to sleep, FlexSC notifies the work-queue, and another thread wakes up to immediately start executing the next system call. Later, when resources become free, current Linux code wakes up the waiting thread (in our case, a syscall thread), and resumes its execution, so it can post its result to the syscall page and return to wait in the FlexSC work-queue.

As previously described, there is never more than one syscall thread concurrently executing per core, for a given process. However in the multicore case, for the same process, there can be as many syscall threads as cores concurrently executing on the entire system. To avoid cache-line contention of syscall pages amongst cores, before a syscall thread begins executing calls from a syscall page, it *locks* the page until all its submitted calls have been issued. Since FlexSC processes typically map multiple syscall pages, each core on the system can schedule a syscall thread to work independently, executing calls from different syscall pages.

4 *Libflexsc*: Asynchronous system call and notification library

To allow event-driven applications to interface with exception-less system calls, we have designed and implemented a simple asynchronous system call notification library, **libflexsc**. Libflexsc provides an event loop for the program, which must register system call requests, along with callback functions. The main event loop on libflexsc invokes the corresponding program provided callback when the system call has completed.

The event loop and callback handling in libflexsc was inspired by the *libevent* asynchronous event notification library [24]. The main difference between these two libraries is that *libevent* is designed to monitor low-level events, such as changes in the availability of input or output, and operates at the file descriptor level. The application is notified of the availability, but its intended operation is still not guaranteed to succeed. For example, a socket may contain available data to be read, but if the application requires more data than is available, it must restate interest in the event to try again. With libflexsc, on the other hand, events correspond to the completion of

```c
conn master;

int main(void)
{
   /* init library and register with kernel */
   flexsc_init();

   /* not performance critical,
                    do synchronously */
   master.fd = bind_and_listen(PORT_NUMBER);

   /* prepare accept */
   master.event->handler = conn_accepted;
   flexsc_accept(&master.event, master.fd,
                 NULL, 0);

   /* jump to event loop */
   return flexsc_main_loop();
}

/* Called when accept() returns */
void conn_accepted(conn *c)
{
   conn *new_conn = alloc_new_conn();

   /* get the return value of the accept() */
   new_conn->fd = c->event->ret;
   new_conn->event->handler = data_read;

   /* issue another accept on the master socket */
   flexsc_accept(&c->event, c->fd, NULL, 0);

   if (new_conn->fd != -1)
      flexsc_read(&new_conn->event, new_conn->fd,
                  new_conn->buf, new_conn->size);
}
```

```c
void data_read(conn *c)
{
   char *reply_file;

   /* read of 0 means connection closed */
   if (c->event->ret == 0) {
       flexsc_close(NULL, c->fd);
       return;
   }

   reply_file = parse_request(c->buf, c->event->ret);

   if (reply_file) {
      c->event->handler = file_opened;
      flexsc_open(&c->event, c->fd, reply_file,
                  O_RDONLY);
   }
}

void file_opened(conn *c)
{
    int file_fd;

    file_fd = c->event->ret;
    c->event->handler = file_sent;
    /* issue asynchronous sendfile */
    flexsc_sendfile(&c->event, c->fd, file_fd,
                    NULL, file_len);
}

void file_sent(conn *c)
{
    /* no callback necessary to handle close */
    flexsc_close(NULL, c->fd);
}
```

Figure 4: Example of network server using libflexsc.

a previously issued exception-less system call. With this model, which is closer to that of asynchronous I/O, it is less likely that applications need to include cumbersome logic to retry incomplete or failed operations.

Contrary to common implementations of asynchronous I/O, FlexSC does not provide a signal or interrupt based completion notification. Completion notification is a mechanism for the kernel to notify a user thread that a previously issued asynchronous request has completed. It is often implemented through a signal or other upcall mechanism. The main reason FlexSC does not offer completion notification is that signals and upcalls entail the same processor performance problems of system calls: direct and indirect processor pollution due to switching between kernel and user execution.

To overcome the lack of completion notifications, the libflexsc event main loop must poll the *syscall pages* currently in use for completion of system calls. To minimize overhead, the polling for system call completion is performed only when all currently pending callback handlers have completed. Given enough work (e.g., handling many connections concurrently), polling should happen infrequently. In the case that all callback handlers have executed, and no new system call has completed, libflexsc falls back on calling `flexsc_wait()` (described in Section 3.2).

4.1 Example server

A simplified implementation of a network server using libflexsc is shown in Figure 4. The program logic is divided into states which are driven by the completion of a previously issued system call. The system calls used in this example that are prefixed with "`flexsc_`" are issued using the exception-less interface (`accept`, `read`, `open`, `sendfile`, `close`). When the library detects the completion of a system call, its corresponding callback handler is invoked, effectively driving the next stage of the state machine. During normal operation, the execution flow of this example would progress in the following order: (1) `main`, (2) `conn_accepted`, (3) `data_read`, (4) `file_opened`, and (5) `file_sent`. As mentioned, file and network descriptors do not need to be marked as non-blocking.

It is worth noting that stages may generate several system call requests. For example, the `conn_accepted()` function not only issues a read on the newly accepted connection, it also issues another accept system call on the master listening socket in order to pipeline further incoming requests. In addition, for improved efficiency, the server may choose to issue *multiple* accepts concurrently (not shown in this example). This would allow the operating system to accept multiple connections without having to first execute user code, as is the case with traditional

event-based systems, thus reducing the number of mode switches for each new connection.

Finally, not all system calls must provide a callback, as a notification may not be of interest to the programmer. For example, in the `file_sent` function listed in the simplified server code, the request to close the file does not provide a callback handler. This may be useful if the completion of a system call does not drive an additional state in the program and the return code of the system call is not of interest.

4.2 Cancellation support

A new feature we had to add to FlexSC in order to support event-based applications is the ability to cancel submitted system calls. Cancellation of in-progress system calls may be necessary in certain cases. For example, servers typically implement a *timeout* feature for reading requests on connections. With non-blocking system calls, reads are implemented by waiting for a notification that the socket has become readable. If the event does not occur within the timeout grace period, the connection is closed. With exception-less system calls, the read request is issued before the server knows if or when new data will arrive (e.g., the `conn_accepted` function in Figure 4). To properly implement a timeout, the application must cancel pending reads if the grace period has passed.

To implement cancellation in FlexSC, we introduced a new *cancel* status value to be used in the status field of the syscall entry (Figure 2). When syscall threads, in the kernel, check for new submitted work, they now also check for entries in *cancel* state. To cancel the in-progress operation, we first identify the syscall thread that is executing the request that corresponds to the cancelled entry. This is easily accomplished since each core has a map of syscall entries to syscall threads for all in-progress system calls. Once identified, a signal is sent to the appropriate syscall thread to interrupt its execution. In the Linux kernel, signal delivery that occurs during system call execution interrupts the system call even if the execution context is asleep (e.g., waiting for I/O). When the syscall thread wakes up, it sets the return value to `EINTR` and marks the entry as *done* in the corresponding syscall entry, after which, the user-mode process knows that the system call has been cancelled and the syscall entry can be reused.

Due to its asynchronous implementation, cancellation requests are not guaranteed to succeed. The window of time between when the application modifies the status field and when the syscall thread is notified of cancellation may be sufficiently long for the system call to complete (successfully). The application must check the system call return code to disambiguate between successly completed calls and cancelled ones. This behavior is analogous to cancellation support of asynchronous I/O implemented by several UNIX systems (e.g., `aio_cancel`).

Server	Total lines of code	Lines of code modifiied	Files modified
memcached	8356	293	3
nginx	82819	255	16

Table 2: Statistics regarding the code size and modifications needed to port applications to libflexsc, measured in lines of code and number of files.

5 Exception-Less Memcached and nginx

This section describes the process of porting two popular event-based servers to use exception-less system calls. In both cases, the applications were modified to conform to the libflexsc interface. However, we strived to maintain the structure of code as similar to the original as possible, to make performance comparisons meaningful.

To reduce the complexity of porting these applications to exception-less system calls, we exploited the fact that FlexSC allows exception-less system calls to co-exist with synchronous ones in the same process. Consequently, we have not modified *all* system calls to use exception-less versions. We focused on the system calls that were issued in the code paths that are involved in handling requests (which correspond to the *hot paths* during normal operation).

5.1 Memcached - Memory Object Cache

Memcached is a distributed memory object caching system, built as an in-memory key-value store [12]. It is typically used to cache results from slower services such as databases and web servers. It is currently used by several popular web sites as a way to improve the performance and scalability of their web services. We used version 1.4.5 as a basis for our port.

To achieve good I/O performance, memcached was built as an event-based server. It uses *libevent* to make use of non-blocking execution available on modern operating system kernels. For this reason, porting memcached to use exception-less system calls through libflexsc was the simplest of the two ports. Table 2 lists the number of lines of code and the number of files that were modified. For memcached, the majority of the changes were done in a single file (`memcached.c`), and the changes were mostly centered around modifying system calls, as well as calls to *libevent*.

Multicore/multiprocessor support has been introduced to memcached, despite most of the code assuming single-threaded execution. To support multiple processors, memcached spawns worker threads which communicate via a pipe to a master thread. The master thread is responsible for accepting incoming connections and handing them to the worker threads.

5.2 nginx Web Server

Nginx is an open-source HTTP web server considered to be light-weight and high-performant; it is currently one of the most widely deployed open-source web servers [26]. Nginx implements I/O concurrency by natively using non-blocking and asynchronous operations available in the operating system kernel. On Linux, nginx uses the `epoll` notification system. We based our port on the 0.9.2 development version of nginx.

Despite having had to change a similar number of lines as with memcached, the port to nginx was more involved, evidenced by the number of files changed (Table 2). This was mainly due to the fact that nginx's core code is significantly larger than that of memcached's (about 10x), and its state machine logic is more complex.

We substituted all system calls that could potentially be invoked while handling client requests to use the corresponding version in libflexsc. The system calls that were associated with a file descriptor based event handler (such as `accept`, `read` and `write`) were straightforward to implement, as these were already programmed as separate stages in the code. However, the system calls that were previously invoked synchronously (e.g., `open`, `fstat`, and `getdents`) needed more work. In most cases, we needed to split a single stage of the state machine into two or more stages to allow asynchronous execution of these system calls. In a few cases, such as `setsockopt` and `close`, we executed the calls asynchronously, without a callback notification, which did not required a new stage in the flow of the program.

Finally, for system calls that not only return a status value, but also fill in a user supplied memory pointer with a data structure, we had to ensure that this memory was correctly managed and passed to the newly created event handler. This requirement prevented the use of stack allocated data structures for exception-less system calls (e.g., programs typically use stack allocated "`struct stat`" data structure to pass to the `fstat` system call).

6 Experimental Evaluation

In this section, we evaluate the performance of exception-less system call support for event-driven servers. We present experimental results of the two previously discussed event-driven servers: memcached and nginx.

FlexSC was implemented in the Linux kernel, version 2.6.33. The baseline measurements were collected using unmodified Linux (same version), with the servers configured to use the `epoll` interface. In the graphs shown, we identify the baseline configuration as "**epoll**", and the system with exception-less system calls as "**flexsc**".

The experiments presented in this section were run on an Intel Nehalem (Core i7) processor with the characteristics shown in Table 3. The processor has 4 cores, each with 2 hyper-threads. We disabled the hyper-threads, as well as the "TurboBoost" feature, for all our experiments to more easily analyze the measurements obtained.

Component	Specification
Cores	4
Cache line	64 B for all caches
Private L1 i-cache	32 KB, 3 cycle latency
Private L1 d-cache	32 KB, 4 cycle latency
Private L2 cache	512 KB, 11 cycle latency
Shared L3 cache	8 MB, 35-40 cycle latency
Memory	250 cycle latency (avg.)
TLB (L1)	64 (data) + 64 (instr.) entries
TLB (L2)	512 entries

Table 3: Characteristics of the 2.3GHz Core i7 processor.

For the experiments involving both servers, requests were generated by a remote client connected to our test machine through a 1 Gbps network, using a dedicated router. The client machine contained a dual core Core2 processor, running the same Linux installation as the test machine.

All values reported in our evaluation represent the average of 5 separate runs.

6.1 Memcached

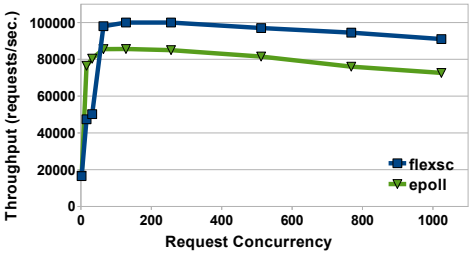
The workload we used to drive memcached is the *memslap* benchmark that is distributed with the libmemcached client library. The benchmark performs a sequence of memcache `get` and `set` operations, using randomly generated keys and data. We configured memslap to issue 10% of set requests and 90% of get requests.

For the baseline experiments (Linux `epoll`), we configured memcached to run with the same number of threads as processor cores, as we experimentally observed this yielded the best baseline performance. For our exception-less version, a single memcached thread was enough to generate enough kernel work to keep all cores busy.

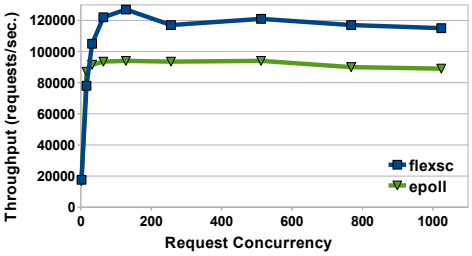
Figure 5 shows the throughput obtained from executing the baseline and exeception-less memcached on 1, 2 and 4 cores. We varied the number of concurrent connections generating requests from 1 to 1024. For the single core experiments, FlexSC employs system call batching, and for the multicore experiments it additionally dynamically distributed system calls to other cores to maximize core locality.

The results show that with 64 or more concurrent requests, memcached programmed to libflexsc outperforms the version using Linux `epoll`. Throughput is improved by as much as 25 to 35%, depending on the number of cores used.

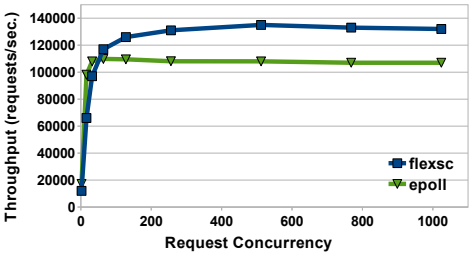
To better understand the source of performance improvement, we collected several performance metrics of the processor using hardware performance counters. Figure 6 shows the effects of executing with FlexSC, while



(a) 1 Core



(b) 2 Cores



(c) 4 Cores

Figure 5: Comparison of Memcached throughput of Linux `epoll` and FlexSC executing on 1, 2 and 4 cores.

servicing 768 concurrent memslap connections. The most important metric listed is the cycles per instruction (CPI) of the user and kernel mode for the different setups, as it summarizes the efficiency of execution (the lower the CPI, the more efficient the execution). The other values listed are normalized values of *misses* on the listed structure (the lower the misses, the more efficient the execution).

The CPI of both kernel and user execution, on 1 and 4 cores, is improved with FlexSC. On a single core, user-mode CPI decreases by as much as 22%, and on the 4 cores, we observe a 52% decrease in user-mode CPI. The data shows that, for memcached, the improved execution comes from significant reduction in misses in the performance sensitive L1, both in the data and instruction part (labelled as *d-cache* and *i-cache*).

The main reason for this drastic increase of user CPI on 4 cores is that with traditional system calls, a user-mode thread must occupy each core to make use of it. With FlexSC, however, if a single user-mode thread generates many system requests, they can be distributed and serviced to remote cores. In this experiment, a single memcached thread was able to generate enough requests to occupy the remaining 3 cores. This way, the core executing the memcached core was predominantly filled with state from the memcached process.

6.2 nginx

To evaluate the effect of exception-less execution of the nginx web server, we used two workloads: ApacheBench and a modified version of httperf. For both workloads, we present results with nginx execution on 1 and 2 cores. The results obtained with 4 cores were not meaningful as the client machine could not keep up with the server, making the client the bottleneck. For the baseline experiments (Linux `epoll`), we configured nginx to spawn one worker process per core, which nginx automatically assigns and pins to separate cores. With FlexSC, a single nginx worker thread was sufficient to keep all cores busy.

6.2.1 ApacheBench

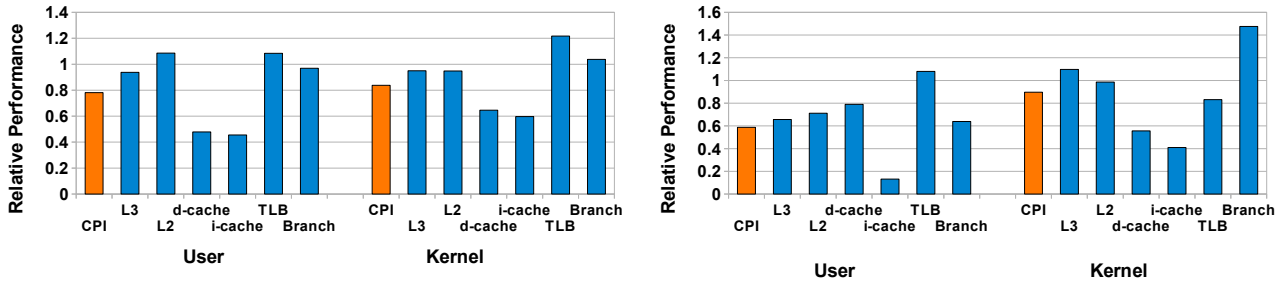
ApacheBench is a HTTP workload generator that is distributed with Apache. It is designed to stress-test the Web server determining the number of requests per second that can be serviced, with varying number of concurrent requests.

Figure 7 shows the throughput numbers obtained on 1 and 2 cores when varying the number of concurrent ApacheBench client connections issuing requests to the nginx server. For this workload, system call batching on one core provides significant performance improvements: up to 70% with 256 concurrent requests. In the 2 core execution, we see that FlexSC provides a consistent improvement with 16 or more concurrent clients, achieving up to 120% higher throughput, showing the added benefit of dynamic core specialization.

Besides aggregate throughput, latency of individual requests is an important metric when evaluating performance of web servers. Figure 8 reports the mean latency, as reported by the client, with 256 concurrent connections. FlexSC reduces latency by 42% in single core execution, and 58% in duo core execution. It is also interesting to note that adding a second core helps to reduce the average latency of servicing requests with FlexSC, which is not the case when using the `epoll` facility.

6.2.2 httperf

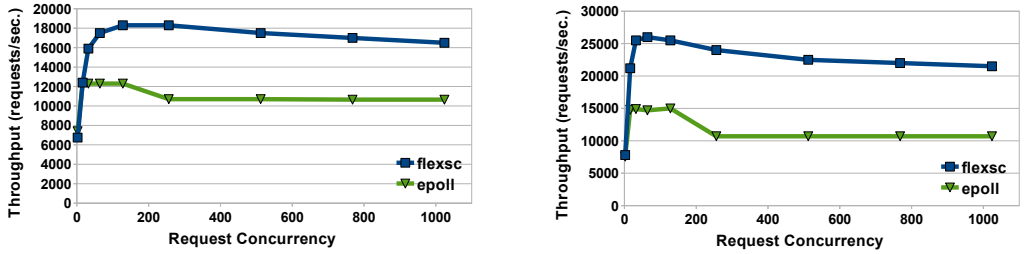
The *httperf* HTTP workload generator was built as a more realistic measurement tool for web server performance [19]. In particular, it supports session log files, and models a *partially open* system (in contrast to ApacheBench, which models a *closed* system) [27]. For this reason, we do not control the number of concurrent



(a) 1 Core

(b) 4 Cores

Figure 6: Comparison of processor performance metrics of Memcached execution using Linux `epoll` and FlexSC on 1 and 4 cores, while servicing 768 concurrent memslap connections. All values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).



(a) 1 Core

(b) 2 Cores

Figure 7: Comparison of nginx performance with the ApacheBench when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

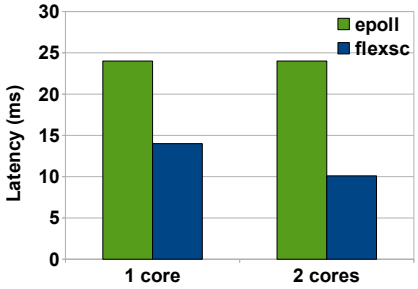


Figure 8: Comparison of nginx latency replying to 256 concurrent ApacheBench requests when executing with Linux `epoll` and FlexSC on 1 and 2 cores.

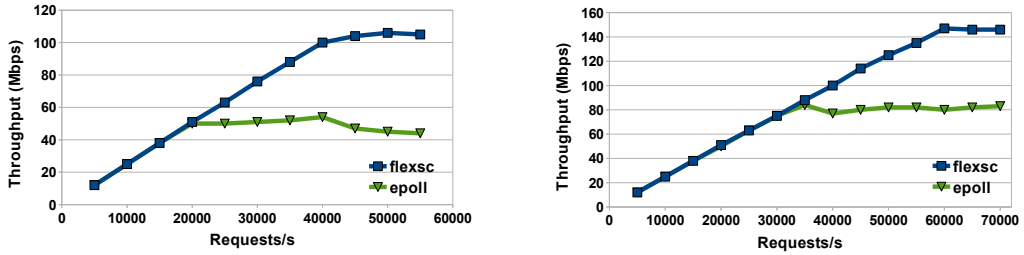
connections to the server, but instead the request arrival rate. The number of concurrent connections is determined by how fast the server can satisfy incoming requests.

We modified httperf (we used the latest version, 0.9.0) in order for it to properly handle large number of concurrent connections. In its original version, httperf uses the `select` system call to manage multiple connections. On Linux, this restricts the number of connections to 1024, which we found insufficient to fully stress the server. We modified httperf to use the `epoll` interface, allowing it to handle several thousand concurrent connections. We verified that the results of our modified httperf were statistically similar to the original httperf, when using less than 1024 concurrent connections.

We configured httperf to connect using HTTP 1.1 protocol, and issue 20 requests per connection. The session log contained requests to files ranging from 64 bytes to 8 kilobytes. We did not add larger files to the session as our network infrastructure is modest, at 1Gpbs, and we did not want the network to become a source of bottleneck.

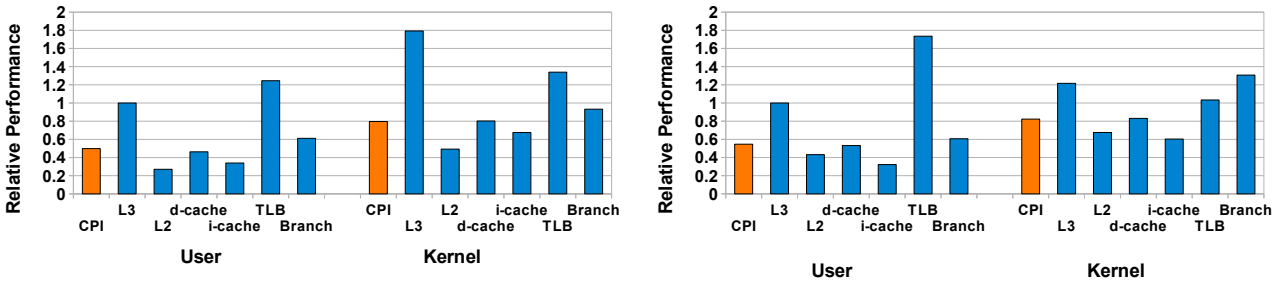
Figure 9 shows the throughput of nginx executing on 1 and 2 cores, measured in megabits per second, obtained when varying the request rate of httperf. Both graphs show that the throughput of the server can satisfy the request rate up to a certain value. After that the throughput is relatively stable and constant. For the single core case, the throughput of Linux `epoll` stabilizes after 20,000 requests per second, while with FlexSC, throughput increases up to 40,000 requests. Furthermore, FlexSC outperforms Linux `epoll` by as much as 120%, when httperf issues 50,000 requests per second.

In the case of 2 core execution, nginx with Linux `epoll` reaches peak throughput at 35,000 requests per second, while FlexSC sustains improvements with up to 60,000 requests per second. In this case, the difference in throughput, in megabits per second, is as much as 77%.



(a) 1 Core (b) 2 Cores

Figure 9: Comparison of nginx performance with the httperf when executing with Linux `epoll` and FlexSC on 1 and 2 cores.



(a) 1 Core (b) 2 Cores

Figure 10: Comparison of processor performance metrics of nginx execution using `epoll` and FlexSC on 1 and 2 cores, while servicing 40,000 and 60,000 req/s, respectively. Values are normalized to baseline execution (`epoll`). The CPI columns show the normalized cycles per instruction, while the other columns depict the normalized misses of each processor structure (lower is better in all cases).

Similarly to the analysis of memcached, we collected processor performance metrics using hardware performance counters to analyze the execution of nginx with httperf. Figure 10 shows the several metrics, normalized to the baselinux (Linux `epoll`) execution. The results show that the efficiency of user-mode execution doubles, in the single core case, and improves by 83% on 2 cores. Kernel-mode execution improves efficiency by 25% and 21%, respectively. For nginx, not only are the L1 instruction and data caches better utilized (we observe less than half of the miss ratio in these structures), but the private L2 cache also observes miss rate reduction of less than half of the baseline.

Although we observe increase of some metrics, such as the TLB and kernel-mode L3 misses, the absolute values are small enough that it does not affect performance significantly. For example, the increase in 80% of kernel-mode L3 misses in the 1 core case corresponds to the misses per kilo instructions increasing from 0.51 to 0.92 (that is, for about every 2,000 instructions, an extra L3 miss is observed). Similarly, the 73% increase in misses of the user-mode TLB in the 2 core execution corresponds to only 4 extra TLB misses for every 1,000 instructions.

The values for the hardware performance information collected during execution driven by ApacheBench showed similar trends (not shown in the interest of space).

7 Discussion: Scaling the Number of Concurrent System Calls

One concern not addressed in this work is that of efficiently handling applications that require a large number of concurrent outstanding system calls. Specifically, there are two issues that can hamper scaling with the number of calls: (1) the exception-less system call interface, and (2) the requirement of one syscall thread per outstanding system call. We briefly discuss mechanisms to overcome or alleviate these issues.

The exception-less system call interface, primarily composed of *syscall entries*, requires user and kernel code to perform linear scans of the entries to search for status updates. If the rate of entry modifications does not grow in same proportion as the total number of entries, the overhead of scanning, normalized per modification, will increase. A concrete example of this is a server servicing a large number of slow or dormant clients, resulting in a large number of connections that are infrequently updated. In this case, requiring linear scans on syscall pages

is inefficient.

Instead of using syscall pages, the exception-less system call interface could be modified to implement two shared message queues: an incoming queue, with system calls requests made by the application, and an outgoing queue, composed of system call requests serviced by the kernel. A queue based interface would potentially complicate user-kernel communication, but would avoid the overheads of linear scans across outstanding requests.

Another scalability factor to consider is the requirement of maintaining a syscall thread per outstanding system call. Despite the modest memory footprint of kernel threads and low overhead of switching threads that share address spaces, these costs may become non-negligible with hundreds of thousands or millions of outstanding system calls.

To avoid these costs, applications may still utilize the `epoll` facility, but through the exception-less interface. This solution, however, would only work for resources that are supported by `epoll`. A more comprehensive solution would be to restructure the Linux kernel to support completely non-blocking kernel code paths. Instead of relying on the ability to block the current context of execution, the kernel could enqueue requests for contended resources, while providing a mechanism to continue the execution of enqueued requests when resources become available. With a non-blocking kernel structure, a single syscall thread would be sufficient to service any number of syscall requests.

One last option to mitigate both the interface and threading issues, that does not involve changes to FlexSC, is to require user-space to throttle the number of outstanding system calls. In our implementation, throttling can be enforced within the libflexsc library by allocating a fixed number of syscall pages, and delaying new system calls whenever all entries are busy. The main drawback of this solution is that, in certain cases, extra care would be necessary to avoid a standstill situation (lack of forward progress).

8 Related Work

8.1 Operating System Support for I/O Concurrency

Over the years, there have been numerous proposals and studies exploring operating system support for I/O concurrency. Due to space constraints, we will briefly describe previous work that is directly related to this proposal.

Perhaps the most influential work in this area is Scheduler Activations that proposed addressing the issue of preempting user-mode threads by returning control of execution to a user-mode scheduler, through a scheduler activation, upon experiencing a blocking event in the kernel [2].

Elmeleegy et al. proposed lazy asynchoronous I/O, a user-level library that uses Scheduler Activations to support event-driven programming [11]. LAIO is the proposal that most closely resembles ours. However, in LAIO, system calls are still exception-based, and tentatively execute synchronously. Since LAIO makes use of scheduler activations, if a blocking condition is detected, a continuation is created, allowing the user thread to continue execution. Recently, the Linux community has proposed a mechanism similar to LAIO for implementing non-blocking system calls [8].

Banga et al. are among the first to explore the construction of generic event notification infrastructure under UNIX [4]. Their work inspired the implementation of the *kqueue* interface available on BSD and Linux kernels [16]. While their proposal does encapsulate more resources than descriptor based ones, explicit kernel support is needed for each type of event. In contrast, exception-less system calls supports all system calls without code specific to each system call or resource.

The main difference between many of the proposals for non-blocking or asynchronous execution and FlexSC is that none of the non-blocking system call proposals completely decouple the invocation of the system call from its execution. As we have discussed, the flexibility resulting from this decoupling is crucial for efficiently exploring optimizations such as system call batching and core specialization.

8.2 Locality of Execution and Multicores

Several researchers have studied the effects of operating system execution on application performance [1, 3, 9, 15, 17]. Larus and Parkes also identified processor inefficiencies of server workloads, although not focusing on the interaction with the operating system. They proposed Cohort Scheduling to efficiently execute staged computations to improve locality of execution [15].

Techniques such as Soft Timers [3] and Lazy Receiver Processing [10] also address the issue of locality of execution, from the other side of the compute stack: handling device interrupts. Both techniques describe how to limit processor interference associated with interrupt handling, while not impacting the latency of servicing requests.

Computation Spreading, proposed by Chakraborty et al., is similar to the multicore execution of FlexSC [9]. They introduced processor modifications to allow for hardware migration of threads, and evaluated the effects on migrating threads to specialize cores when they enter the kernel. Their simulation-based results show an improvement of up to 20% on Apache; however, they explicitly do not model TLBs and provide for fast thread migration between cores. On current hardware, synchronous thread migration between cores requires a costly interprocessor interrupt.

Recently, both Corey and Factored Operating System (*fos*) have proposed dedicating cores for specific operating system functionality [31, 32]. There are two main differences between the core specialization possible with these proposals and FlexSC. First, both Corey and *fos* require a micro-kernel design of the operating system kernel in order to execute specific kernel functionality on dedicated cores. Second, FlexSC can dynamically adapt the proportion of cores used by the kernel, or cores shared by user and kernel execution, depending on the current workload behavior.

Explicit off-loading of select OS functionality to cores has also been studied for performance [20, 21] and power reduction in the presence of single-ISA heterogeneous multicores [18]. While these proposals rely on expensive inter-processor interrupts to offload system calls, we hope FlexSC can provide for a more efficient and flexible mechanism that can be used by such proposals.

Zeldovich et al. introduced *libasync-smp*, a library that allows event-driven servers to execute on multiprocessors by having programmers specify events that can be safely handled concurrently [33]. *Libasync-smp* was designed to use existing asynchronous I/O facilities of UNIX kernels, but could be extended to rely on exception-less system calls instead.

8.3 System Call Batching

The idea of batching calls in order to save crossings has been extensively explored in the systems community. Closely related to this work is the work by Bershad et al. on user-level remote procedure calls (URPC) [6]. In particular, the use of shared memory to communicate requests, allied with the use of light-weight threads is common in both URPC and FlexSC. In this work, we explored directly *exposing* the communication mechanism to the application thereby removing the reliance on user-level threads.

Also related to exception-less system calls are *multi-calls*, which are used in both operating systems and paravirtualized hypervisors as a mechanism to address the high overhead of mode switching. Cassyopia is a compiler targeted at rewriting programs to collect many independent system calls, and submitting them as a single multi-call [25]. An interesting technique in Cassyopia, which could be eventually explored in conjunction with FlexSC, is the concept of a *looped multi-call* where the result of one system call can be automatically fed as an argument to another system call in the same multi-call. In the context of hypervisors, both Xen and VMware currently support a special multi-call hypercall feature [5][29].

9 Concluding Remarks

Event-driven architectures continue to be a popular design option for implementing high-performance and scalable server applications. This paper proposes the use of *exception-less system calls* as the principal operating system primitive for efficiently supporting I/O concurrency and event-driven execution. We describe several advantages of exception-less system calls over traditional support for I/O concurrency and event notification facilities, including: (1) any system call can be invoked asynchronously, even system calls that are not file descriptor-based, (2) support in the operating system kernel is non-intrusive as code changes are not required to each system call, (3) processor efficiency is high since mode switches are mostly avoided when issuing or executing asynchronous operations, and (4) enabling multi-core execution for event-driven programs is easier, given that a single user-mode execution context can generate a sufficient number of requests to keep multiple processors/-cores busy with kernel execution.

We described the design and implementation of *libflexsc*, an asynchronous system call and notification library that makes use of our Linux exception-less system call implementation, called FlexSC. We show how libflexsc can be used to support current event-driven servers by porting two popular server applications to the exception-less execution model: memcached and nginx.

The experimental evaluation of libflexsc demonstrates that the proposed exception-less execution model can significantly improve the performance and efficiency of event-driven servers. Specifically, we observed that exception-less execution increases the throughput of memcached by up to 35%, and that of nginx by up to 120%. We show that the improvements, in both cases, are derived from more efficient execution through improved use of processor resources.

10 Acknowledgements

This work was supported in part by Discovery Grant funding from the Natural Sciences and Engineering Research Council (NSERC) of Canada. We would like to thank the valuable feedback from the reviewers and our shepherd, Richard West. Special thanks to Ioana Burcea for innumerous discussions that influenced the development of this work.

References

[1] AGARWAL, A., HENNESSY, J., AND HOROWITZ, M. Cache performance of operating system and multiprogramming workloads. *ACM Trans. Comput. Syst. (TOCS) 6*, 4 (1988), 393–431.

[2] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler Activations: Effective Kernel Support for

the User-Level Management of Parallelism. *ACM Trans. Comput. Syst. 10*, 1 (1992), 53–79.

[3] ARON, M., AND DRUSCHEL, P. Soft timers: efficient microsecond software timer support for network processing. *ACM Trans. Comput. Syst. (TOCS) 18*, 3 (2000), 197–228.

[4] BANGA, G., MOGUL, J. C., AND DRUSCHEL, P. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), USENIX Association, pp. 19–19.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[6] BERSHAD, B. N., ANDERSON, T. E., LAZOWSKA, E. D., AND LEVY, H. M. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst. 9* (May 1991), 175–198.

[7] BHATTACHARYA, S., PRATT, S., PULAVARTY, B., , AND MORGAN, J. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Ottawa Linux Symposium* (2003), pp. 371–386.

[8] BROWN, Z. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)* (2007), pp. 81–85.

[9] CHAKRABORTY, K., WELLS, P. M., AND SOHI, G. S. Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2006), pp. 283–292.

[10] DRUSCHEL, P., AND BANGA, G. Lazy receiver processing (LRP): a network subsystem architecture for server systems. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (1996), pp. 261–275.

[11] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2004), pp. 21–21.

[12] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal* (2004).

[13] GAMMO, L., BRECHT, T., SHUKLA, A., AND PARIAG, D. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of 6th Annual Ottawa Linux Symposium* (2004), pp. 215–225.

[14] KROHN, M., KOHLER, E., AND KAASHOEK, M. F. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), USENIX Association, pp. 7:1–7:14.

[15] LARUS, J., AND PARKES, M. Using Cohort-Scheduling to Enhance Server Performance. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC)* (2002), pp. 103–114.

[16] LEMON, J. Kqueue: A generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 141–153.

[17] MOGUL, J. C., AND BORG, A. The Effect of Context Switches on Cache Performance. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1991), pp. 75–84.

[18] MOGUL, J. C., MUDIGONDA, J., BINKERT, N., RANGANATHAN, P., AND TALWAR, V. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro 28*, 3 (2008), 26–41.

[19] MOSBERGER, D., AND JIN, T. httperf – A Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev. 26* (December 1998), 31–37.

[20] NELLANS, D., BALASUBRAMONIAN, R., AND BRUNVAND, E. OS execution on multi-cores: is out-sourcing worthwhile? *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 104–105.

[21] NELLANS, D., SUDAN, K., BRUNVAND, E., AND BALASUBRAMONIAN, R. Improving Server Performance on Multi-Cores via Selective Off-loading of OS Functionality. In *Sixth Annual Workshorp on the Interaction between Operating Systems and Computer Architecture (WIOSCA)* (2010), pp. 13–20.

[22] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: an efficient and portable web server. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1999), USENIX Association, pp. 15–15.

[23] PARIAG, D., BRECHT, T., HARJI, A., BUHR, P., SHUKLA, A., AND CHERITON, D. R. Comparing the performance of Web server architectures. In *Proceedings of the 2nd European Conference on Computer Systems (Eurosys)* (2007), pp. 231–243.

[24] PROVOS, N. libevent - An Event Notification Library. http://www.monkey.org/~provos/libevent.

[25] RAJAGOPALAN, M., DEBRAY, S. K., HILTUNEN, M. A., AND SCHLICHTING, R. D. Cassyopia: compiler assisted system optimization. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)* (2003), pp. 18–18.

[26] REESE, W. Nginx: the High-Performance Web Server and Reverse Proxy. *Linux Journal* (2008).

[27] SCHROEDER, B., WIERMAN, A., AND HARCHOL-BALTER, M. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, pp. 18–18.

[28] SOARES, L., AND STUMM, M. Flexsc: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010), pp. 33–46.

[29] VMWARE. *VMWare Virtual Machine Interface Specification*. http://www.vmware.com/pdf/vmi_specs.pdf.

[30] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 230–243.

[31] WENTZLAFF, D., AND AGARWAL, A. Factored Operating Systems (fos): The Case for a Scalable Operating System for Multicores. *SIGOPS Oper. Syst. Rev. 43*, 2 (2009), 76–85.

[32] WICKIZER, S. B., CHEN, H., CHEN, R., MAO, Y., KAASHOEK, F., MORRIS, R., PESTEREV, A., STEIN, L., WU, M., DAI, Y., ZHANG, Y., AND ZHANG, Z. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2008).

[33] ZELDOVICH, N., YIP, A., DABEK, F., MORRIS, R., MAZIÈRES, D., AND KAASHOEK, F. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX)* (June 2003).

Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming

Josh Triplett
Portland State University
josh@joshtriplett.org

Paul E. McKenney
IBM Linux Technology Center
paulmck@linux.vnet.ibm.com

Jonathan Walpole
Portland State University
walpole@cs.pdx.edu

Abstract

We present algorithms for shrinking and expanding a hash table while allowing concurrent, wait-free, linearly scalable lookups. These resize algorithms allow Read-Copy Update (RCU) hash tables to maintain constant-time performance as the number of entries grows, and reclaim memory as the number of entries decreases, without delaying or disrupting readers. We call the resulting data structure a *relativistic hash table*.

Benchmarks of relativistic hash tables in the Linux kernel show that lookup scalability during resize improves 125x over reader-writer locking, and 56% over Linux's current state of the art. Relativistic hash lookups experience no performance degradation during a resize. Applying this algorithm to memcached removes a scalability limit for get requests, allowing memcached to scale linearly and service up to 46% more requests per second.

Relativistic hash tables demonstrate the promise of a new concurrent programming methodology known as *relativistic programming*. Relativistic programming makes novel use of existing RCU synchronization primitives, namely the *wait-for-readers* operation that waits for unfinished readers to complete. This operation, conventionally used to handle reclamation, here allows ordering of updates without read-side synchronization or memory barriers.

1 Introduction

Hash tables offer applications and operating systems many convenient properties, including constant average time for accesses and modifications [3, 10]. Hash tables used in concurrent applications require some sort of synchronization to maintain internal consistency. Frequently accessed hash tables will become application bottlenecks unless this synchronization scales to many threads on many processors.

Existing concurrent hash tables primarily make use of mutual exclusion, in the form of locks. These approaches do not scale, due to contention for those locks. Alternative implementations exist, using non-blocking synchronization or transactions, but many of these techniques still require expensive synchronization operations, and still do not scale well. Running any of these hash-table implementations on additional processors does not provide a proportional increase in performance.

One solution for scalable concurrent hash tables comes in the form of Read-Copy Update (RCU) [18, 16, 12]. Read-Copy Update provides a synchronization mechanism for concurrent programs, with very low overhead for readers [13]. Thus, RCU works particularly well for data structures with significantly more reads than writes; this category includes many data structures commonly used in operating systems and applications, such as read-mostly hash tables.

Existing RCU-based hash tables use open chaining, with RCU-based linked lists for each hash bucket. These tables support insertion, removal, and lookup operations [13]. Our previous work introduced an algorithm to move hash items between hash buckets due to a change in the key [24, 23], making RCU-based hash tables even more broadly usable.

Unfortunately, RCU-based hash tables still have a major deficiency: they do not support *dynamic resizing*.

The performance of a hash table depends heavily on the number of hash buckets. Making a hash table too small will lead to excessively long hash chains and poor performance. Making a hash table too large will consume too much memory, increasing hardware requirements or reducing the memory available for other applications or performance-improving caches. Many users of hash tables cannot know the proper size of a hash table in advance, since no fixed size suits all system configurations and workloads, and the system's needs may change at runtime. Such systems require a hash table that supports dynamic resizing.

Resizing a concurrent hash table based on mutual exclusion requires relatively little work: simply acquire the appropriate locks to exclude concurrent reads and writes, then move items to a new table. However, RCU-based hash tables *cannot exclude readers*. This property proves critical to RCU's scalability and performance, since excluding readers would require expensive read-side synchronization. Thus, any RCU-based hash-table resize algorithm must cope with concurrent reads while resizing.

Solving this problem without reducing read performance has seemed intractable. Existing RCU-based scal-

able concurrent hash tables in the Linux kernel, such as the directory-entry cache (dcache) [17, 11], do not support resizing; they allocate a fixed-size table at boot time based on system heuristics such as available memory. Nick Piggin proposed a resizing algorithm for RCU-based hash tables, known as "Dynamic Dynamic Data Structures" (DDDS) [21], but this algorithm slows common-case lookups by requiring them to check multiple hash tables, and this slowdown increases significantly during a resize.

As our primary contribution, we present the first algorithm for resizing an RCU-based hash table without blocking or slowing concurrent lookups. Because lookups can occur at any time, we keep our *relativistic* hash table in a consistent state at all times, and never allow a lookup to spuriously miss an entry due to a concurrent resize operation. Furthermore, our resize algorithms avoid copying the individual hash-table nodes, allowing readers to maintain persistent references to table entries.

A key insight made our relativistic hash table possible. We use an existing RCU synchronization primitive, the *wait-for-readers* operation, to control which versions of the hash-table data structure concurrent readers can observe. This use of wait-for-readers generalizes and subsumes its original application of safely managing memory reclamation. This general-purpose ordering primitive forms the basis of a new concurrent programming methodology, which we call *relativistic programming* (RP). Relativistic programming enables scalable, high-performance data structures previously considered intractable for RCU.

We use the phrase *relativistic programming* by analogy with relativity, in which observers can disagree on the order of causally unrelated events. Relativistic programming aims to minimize synchronization, by allowing reader operations to occur concurrently with writers; writers may never block readers to enforce a system-wide serialization of memory operations. Inevitably, then, independent readers can disagree on the order of unrelated writer operations, such as the insertion order of two items into separate hash-table chains; however, writers can still synchronize to preserve the order of causally related operations. Whereas concurrent programming methodologies such as transactional memory always preserve the ordering of even unrelated writes—at significant cost to performance and scalability, since readers must use synchronization to support blocking or retries—relativistic programming provides the means to program even complex, whole-data-structure operations such as resizing with excellent performance and scalability.

Section 2 compares our algorithms to other related work. Section 3 provides an introduction to RCU and to the relativistic programming techniques supporting this work. Section 4 documents our new hash-table resize algorithms, and the corresponding lookup operation. Section 5 describes the other hash-table implementations we tested for comparison. Section 6 discusses the implementation and benchmarking of our relativistic hash-table algorithm, including both microbenchmarks and real-world benchmarks. Section 7 presents and analyzes the benchmark results. Section 8 discusses the future of the relativistic programming methodology supporting this work.

2 Related Work

Relativistic hash tables use the RCU wait-for-readers operation to enforce the ordering and visibility of write operations, without requiring synchronization operations in the reader. This novel use of wait-for-readers evolved through a series of increasingly sophisticated write-side barriers. Paul McKenney originally proposed the elimination of read memory barriers by introducing a new write memory barrier primitive that forced a barrier on all CPUs via inter-processor interrupts [14]. McKenney's later work on Sleepable Read-Copy Update (SRCU) [15] used the RCU wait-for-readers operation to manage the order in which write operations became visible to readers, providing the first example of using wait-for-readers to order non-reclamation operations; this use served the same function as a write memory barrier, but without requiring a corresponding read memory barrier in every RCU reader. Philip Howard further refined this approach in his work on relativistic red-black trees [9], using the wait-for-readers operation to order the visibility of tree rotation and balancing operations and prevent readers from observing inconsistent states. Howard's work used wait-for-readers as a stronger barrier than a write memory barrier, enforcing the order of write operations regardless of the order a reader encounters them in the data structure. Relativistic programming builds on this stronger barrier.

Relativistic and RCU-based data structures typically use mutual exclusion to synchronize between writers. Philip Howard and Jonathan Walpole [8] demonstrated an alternative approach, combining relativistic readers with software transactional memory (STM) writers, and integrating the wait-for-readers operation into the transaction commit. This approach provides scalable high-performance relativistic readers, while also allowing scalable writers within the limits of STM. Relativistic transactions could substantially simplify relativistic hash-table writers compared to fine-grained locking, while still providing good scalability.

Prior attempts to build resizable RCU hash tables have arisen from the limitations of fixed-size RCU hash tables in the Linux kernel. Nick Piggin's DDDS [21] supports hash-table resizes, but DDDS slows down all lookups by

requiring checks for concurrent resizes, and furthermore requires that lookups during resizes examine both the old and the new structures; relativistic hash tables do neither. We discuss DDDS further in section 5. Herbert Xu implemented a resizable multi-hash-table structure based on RCU, in which every hash-table entry contains two sets of linked-list pointers so it can appear in the old and new hash tables simultaneously [25]. Together with a global version number for the structure, this allows readers to effectively snapshot all links in the hash table simultaneously. However, this approach drastically increases memory usage and cache footprint.

Various authors [7, 5, 19, 2] have proposed resizable concurrent hash tables. Unlike relativistic hash tables, these algorithms require expensive synchronization operations on reads, such as locks, atomic instructions, or memory barriers. Furthermore, like DDDS, several of these algorithms require retries on failure.

Maurice Herlihy and Nir Shavit documented numerous concurrent hash tables, including both open-chained and closed tables [7]; all of these require expensive synchronization, and some require retries. Gao, Groote, and Hesselink proposed a lock-free hash table using closed hashing [5]; their approach relies on atomic operations and on helping concurrent operations complete.

Maged Michael implemented a lock-free hash table based on compare and swap (CAS) [19], though he did not propose a resize algorithm. Michael's table lookups avoid most expensive synchronization operations in the common case (with the exception of read barriers), but must retry on any concurrent modification. To support safe memory reclamation, Michael uses hazard pointers [20], which provide a wait-for-readers operation similar to that of RCU; hazard pointers can reduce wait-for-readers latency, but impose higher reader cost [6].

Relativistic hash tables use open hashing with per-bucket chaining. Closed hash tables, which store entries inline in the array, can offer smaller lookup cost and better cache behavior, but force copies on resize. Closed tables also require more frequent resizing, as they do not gracefully degrade in performance when overloaded, but rather become pathologically more expensive and then stop working entirely. Depending on the implementation, removals from the table may not make the table any emptier, as the entries must remain as "tombstones" to preserve reader probing behavior.

Cliff Click presented a scalable lock-free resizable hash for Java based on closed hashing [2]; this hash avoids most synchronization operations for readers and writers by leaving the ordering of memory operations entirely unspecified and reasoning about all possible resulting memory states and transitions. (Readers require a read memory barrier but no other synchronization. Writers require a CAS but not a write memory barrier.) Click's use of state-based reasoning to avoid ordering provides an interesting and potentially higher-performance alternative to the causal-order enforcement in relativistic writers. In contrast with relativistic hash tables, but like DDDS, Click's hash-table readers must probe alternate hash tables during resizing.

Other approaches to resizable hash tables include that of Ori Shalev and Nir Shavit, who proposed a "split-ordered list" structure consisting of a single linked list with hash buckets pointing to intermediate list nodes [22, 7]. This structure allows resizing by adding or removing buckets, splitting or joining the existing buckets respectively. This approach keeps the underlying linked list in a novel sort order based on the hash key, as with the variation of our algorithms proposed in section 4.3, to allow splitting or joining buckets without reordering. Split-ordered lists seem highly amenable to a simple relativistic implementation, making the lookups scalable and synchronization-free while preserving the lock-free modifications and simple resizes; we plan to implement a relativistic split-ordered list in future work.

Our previous work developed a relativistic algorithm for moving a hash-table entry from one bucket to another atomically [24, 23]. This algorithm introduced the notion of cross-linking hash buckets to make entries appear in multiple buckets simultaneously. However, this move algorithm required changing the hash key and potentially copying the entry.

We chose to implement our benchmarking framework `rcuhashbash-resize` as a Linux kernel module, as documented in section 6.1. However, several portable RCU implementations exist outside the Linux kernel. Mathieu Desnoyers reimplemented RCU as a POSIX userspace library, liburcu, for use with pthreads, with no Linux-specific code outside of optional optimizations [4]. For our real-world benchmarks with the memcached key-value storage engine (documented in section 6.2), we used liburcu to support our modified storage engine.

3 Read-Copy Update Background

Read-Copy Update (RCU) provides synchronization between readers and writers of a shared data structure. In sharp contrast to locking, non-blocking synchronization, or transactional memory, RCU readers perform *no expensive synchronization operations whatsoever*: no locks, no atomic operations, no compare-and-swap, and no memory barriers. RCU readers typically incur little to no overhead even compared to concurrency-unsafe single-threaded implementations; furthermore, by avoiding expensive synchronization, RCU readers avoid the need for communication between threads, allowing wait-free operation and excellent scalability.

RCU readers execute concurrently, both with each

other and with writers, and thus readers can potentially observe writers in progress. (Other concurrent programming models prevent readers from viewing intermediate memory states via locking or conflict detection.) The methodologies of RCU-based concurrent programming primarily address the safe management of reader/writer concurrency. Since writers may not impede readers in any way, programmers must reason about the memory states readers can observe, and avoid exposing inconsistent intermediate states from writers.

Commonly, writers preserve data-structure invariants by atomically transitioning data structures between consistent states. On all existing CPU architectures, aligned writes to machine-word-sized memory regions (such as pointers) have atomic semantics, such that a reader sees either the old or the new state, with no intermediate value; thus, structures linked together via pointers support many structural manipulations via direct updates. For more complex manipulations, such as insertion of a new item into a data structure, RCU writers typically allocate memory initially unreachable by readers, initialize it, and then atomically *publish* it by updating a pointer in reachable memory. The publish operation requires a write memory barrier between initialization and publication, to ensure that readers traversing the pointer will observe initialized memory. Readers may also require compiler directives to prevent certain aggressive optimizations across the pointer dereference; RCU wraps those directives into a *read* primitive.[1]

These operations allow RCU writers to update data structures and maintain invariants for readers. However, RCU writers must also manage object lifetimes, which requires knowing when readers might hold references to an item in memory. Unlinking an item from a data structure makes it unreachable to new readers, but does not stop accesses from unfinished readers; writers may not reclaim the unlinked item's memory until all such readers have completed. This resembles a garbage collection problem, but RCU must support runtime environments without automatic garbage collection.

To this end, RCU provides a barrier-like synchronization operation called *wait-for-readers*, which blocks until all readers which started before the barrier have completed. Thus, once a writer makes memory unreachable from the published data structure, a wait-for-readers operation ensures that no readers still hold references to that memory. Wait-for-readers does not prevent *new* readers from starting; it simply waits for existing *unfinished* readers to complete. This barrier operates conservatively: the currently unfinished readers might not hold references to that item, and the barrier itself may wait longer than strictly necessary in order to run efficiently or batch several reclamations into a single wait operation. This conservative semantic allows much more efficient and scalable implementations, particularly for readers. The portion of a writer that follows a wait-for-readers barrier often consists only of memory reclamation; because memory-reclamation operations can safely occur concurrently and need not occur immediately (assuming sufficient memory), common RCU APIs also provide an asynchronous wait-for-readers callback.

However, writers have more reasons to order operations than just reclaiming memory. Our work on relativistic programming provides a general framework for writers to enforce the ordering of operations visible to readers, using the same wait-for-readers primitive. Relativistic programming builds on RCU's key benefits—minimized communication, minimized expensive synchronization, and readers that run concurrently with writers—for scalability. We present here a specific application of the relativistic programming methodology to maintain the data-structure invariants of a hash table while resizing it; section 8 discusses the future development of the general methodology to support maintenance of arbitrary data-structure invariants.

4 Relativistic Hash Tables

Any hash table requires a hash function, which maps entries to hash buckets based on their key. The same key will always hash to the same bucket; different keys will ideally hash to different buckets, but may map to the same bucket, requiring some kind of conflict resolution. The algorithms described here work with hash tables using *open chaining*, where each hash bucket has a linked list of entries whose keys hash to that bucket. As the number of entries in the hash table grows, the average depth of a bucket's list grows and lookups become less efficient, necessitating a resize.

Resizing the table requires allocating a new region of memory for the new number of hash buckets, then linking all the nodes into the new buckets. To allow resizes to atomically substitute the new hash table for the old, readers access the hash-table structure through a pointer; this structure includes the array of buckets and the size of the table.

Our resize algorithms synchronize with the corresponding lookup algorithm using existing RCU programming primitives. However, any semantically equivalent implementation will work.

The **RP hash lookup reader** follows the standard algorithm for open-chain hash table lookups:

1. Snapshot the hash-table pointer in case a resizer replaces it during the lookup.

[1] On certain obsolete architectures (DEC Alpha), readers must also use a memory barrier.

2. Hash the desired key, modulo the number of buckets.
3. Look up the corresponding hash bucket in the array.
4. Traverse the linked list, comparing each entry's key to the desired key.
5. If the current entry's key matches the desired key, the desired value appears in the same entry; use or return that value.[2]

In a concrete implementation, lookup (like any RCU reader) will include explicit operations delimiting the start and end of the reader. Depending on the choice of RCU implementation, these delimiter operations may compile to compiler directives, to requests to prevent preemption, to manipulation of CPU-local or thread-local counters, or to other lightweight operations.

Lookups will traverse the hash table concurrently with other operations, including resizes. To avoid disrupting lookups, we require that a lookup can never fail to find a node, even in the presence of a concurrent resize. This means that each hash chain must contain those items that hash to the corresponding bucket. Most prior hash table resize algorithms ensure that a hash chain contains *exactly* those items. We loosen this constraint, instead allowing hash chains to ephemerally contain items that hash to *different* buckets. We call such hash chains *imprecise* since they include all items which hash to that bucket but may include others as well. Readers and writers must tolerate imprecise hash chains (although some operations, such as lookup, require no adaptation). Imprecise hash chains allow us to resize hash tables and otherwise manipulate buckets without copying items or wasting memory.

For simplicity, relativistic hash tables constrain resizing to change the number of buckets by integral factors—for instance, doubling or halving the number of buckets. This guarantees two constraints: First, when shrinking the table, each bucket of the new table will contain all entries from multiple buckets of the old table; and second, when growing the table, each bucket of the new table will contain entries from at most one bucket of the old table.

Based on the first constraint, the **RP hash shrink writer** can shrink a table as follows:

1. Allocate the new, smaller table.
2. Link each bucket in the new table to the first bucket in the old table that contains entries which will hash to the new bucket.
3. Link the end of each such bucket to the beginning of the next such bucket; each new bucket will thus chain through as many old buckets as the resize factor.
4. Set the table size.
5. Publish the new, valid hash table.
6. Wait for readers. No new readers will have references to the old hash table.
7. Reclaim the old hash table.

Concurrent inserts and removes must block until the shrink algorithm finishes publishing the new hash table and waits for readers to drop references to the old table. See section 4.1 for further details on insertion and removal.

For an example of the shrink algorithm, see figure 1.

Based on the second constraint, the **RP hash expand writer** can expand a table as follows:

1. Allocate the new, larger table.
2. For each new bucket, search the corresponding old bucket for the first entry that hashes to the new bucket, and link the new bucket to that entry. Since all the entries which will end up in the new bucket appear in the same old bucket, this constructs an entirely valid new hash table, but with multiple buckets "zipped" together into a single imprecise chain.
3. Set the table size.
4. Publish the new table pointer. Lookups may now traverse the new table, but they will not benefit from any additional efficiency until later steps unzip the buckets.
5. Wait for readers. All new readers will see the new table, and thus no references to the old table will remain.
6. For each bucket in the old table (each of which contains items from multiple buckets of the new table):
   - 6.1 Advance the old bucket pointer one or more times until it reaches a node that doesn't hash to the same bucket as the previous node. Call the previous node p.
   - 6.2 Find the subsequent node which does hash to the same bucket as node p, or NULL if no such node exists.
   - 6.3 Set p's next pointer to that subsequent node pointer, bypassing the nodes which do not hash to p's bucket.
7. Wait for readers. New readers will see the changes made in this pass, so they won't miss a node during the next pass.

[2] If the lookup algorithm needs to hold a reference to the entry after the reader ends, it must take any additional steps to protect that entry before ending the reader.

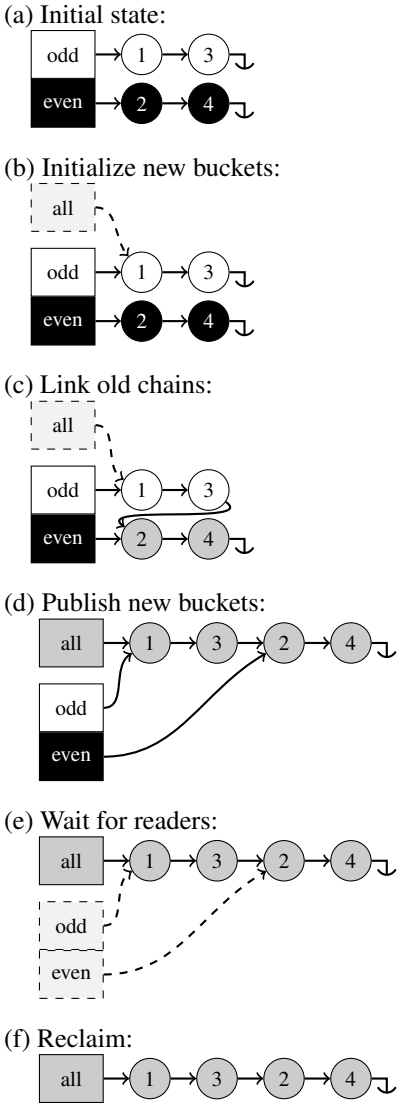


Figure 1: Shrinking a relativistic hash table. (a) The initial state has two buckets, one for odd numbers and one for even numbers. White nodes indicate reachability by odd readers, and black nodes by even readers. (b) The resizer allocates a new one-bucket table and links it to the appropriate old bucket. Dashed nodes exist only in writer-private memory, unreachable by readers. (c) The resizer links the odd bucket's chain to the even bucket, making the odd bucket's chain imprecise. Gray nodes indicates reachability by both odd and even readers. (d) The resizer publishes the new table. (e) After waiting for readers, (f) the resizer can free the old table.

8. If any changes occurred in this pass, repeat from step 6. Note that this loop depends only on the interleaving of nodes with different destination buckets in the zipped bucket, not on subsequent inserts or removals; thus, this loop cannot livelock.
9. Reclaim the old hash table.

The wait in step 7 orders unzip operations for concurrent readers. Without it, a reader traversing a zipped chain could follow an updated pointer from an item in a different bucket, and thus erroneously skip some items from its own bucket.

For an example of the expansion algorithm, see figure 2.

This version of the algorithm uses the old hash table for auxiliary storage during unzip steps. The algorithm could avoid this auxiliary storage at the cost of additional traversals.

Concurrent inserts and removes on a given bucket must block until after all unzips have completed on that bucket.

4.1 Handling Insertion and Removal

Existing RCU-based hash tables synchronize insertion and removal operations with concurrent lookups via standard RCU linked-list operations on the appropriate buckets. Multiple insertion and removal operations synchronize with each other using per-bucket mutual exclusion. (Herlihy and Shavit describe a common workload for hash tables as 90% lookups, 9% insertions, and 1% removals [7], justifying an emphasis on fast concurrent lookups. Nevertheless, several other hash table implementations offer finer-grained update algorithms based on compare-and-swap [2, 22], which we could potentially adapt to improve concurrent update performance.) Resizes, however, introduce an additional operation that modifies the hash table, and thus require synchronization with insertions and removals. We initially consider it sufficient to minimize performance degradation versus a non-resizable hash table, particularly with no concurrent resize running.

During the initial period of initializing new buckets and publishing the new table, our resizers block all updates, either using a hash-table-wide reader-writer lock (where inserts and removes acquire a read lock and resizers acquire the write lock) or by acquiring all per-bucket locks [7]. In the simplest case, concurrent updates can continue to block until the resize completes; however, concurrent updates can potentially run earlier if they carefully handle the intermediate states produced by the resizer. For a sufficiently large hash table, this may prove necessary to avoid excessive delays on concurrent updates.

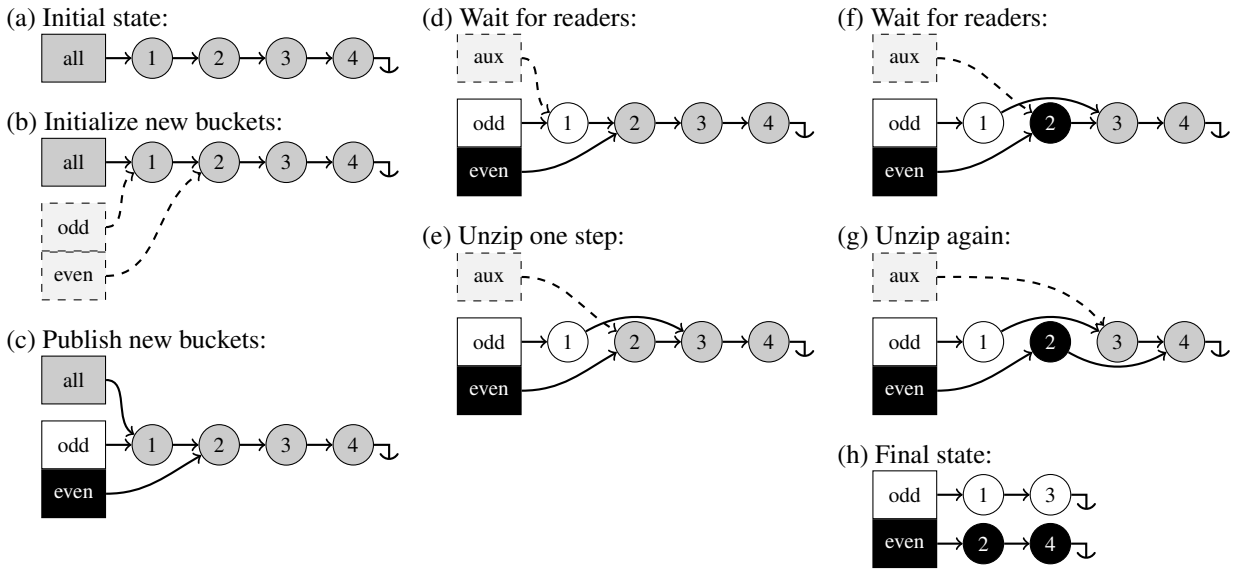


Figure 2: Growing a relativistic hash table. Colors as in figure 1. (a) The initial state contains one bucket. (b) The resizer allocates a new two-bucket table and points each bucket to the first item with a matching hash; this produces valid imprecise hash chains. (c) The resizer can now publish the new hash table. However, an even reader might have read the old hash chain just before publication, making item 1 gray—reachable by both odd and even readers—and preventing safe modification of its next pointer. (d) The resizer waits for readers; new even readers cannot reach item 1. (e) The resizer updates item 1's next pointer to point to the next odd item. (f) After another wait for readers, (g) the unzipping process can continue. (h) The final state.

In our shrink algorithm, the resizer must complete all cross-link steps before publishing the table; once the resizer has published the table, the algorithm has effectively completed with the exception of reclamation, allowing no opportunity for concurrent updates. However, the shrink algorithm could choose to publish the initialized table for updaters as soon as it completes initialization, allowing concurrent updates to proceed while the cross-linking continues. The shrink algorithm may then drop the per-bucket lock for a bucket as soon as it has finished cross-linking that bucket, allowing concurrent insertions and removals on that bucket.

Insertion and removal operations during expansion must take extra care when operating on the zipped buckets. When performing a single unzip pass on a given set of buckets, the expansion algorithm must acquire the per-bucket locks for all buckets in that set. This proves sufficient to handle insertions, which simply insert at the beginning of the appropriate new bucket without disrupting the next resize pass.

Removal, however, may occur at any point in a zipped bucket, including at the location of the resizer's *aux* pointer that marks the start of the next unzip pass. If a removal occurs with a table expansion in progress, the removal must check for a conflict with this aux pointer, and update the pointer if it points to the removed node. Given the relatively low frequency of removal versus lookup and insertion, and the even lower frequency of resizes, we consider it acceptable to require this additional check in the removal algorithm.

4.2 Variation: Resizing in Place

The preceding descriptions of the resize algorithms assumed an out-of-place resize: allocate a new table, move all the nodes, reclaim the old table. However, given a memory allocator which can resize existing allocations without moving them, we can adapt the resize algorithms to resize in place. This has two primary side effects: the resizer cannot count on the new table remaining private until published, and the buckets shared with the old table will remain initialized to the same values.

To shrink a hash table in place, we adapt the previous shrink algorithm to avoid disrupting unfinished readers:

1. The smaller table will consist of a prefix of the current table, and the buckets in that prefix already point to the first of the lists that will appear in those buckets.
2. As before, concatenate all the buckets which contain entries that hash to the same bucket in the smaller table.

3. Wait for readers. All new readers will see the concatenated buckets.
4. Set the table size to the new, smaller size.
5. Wait for readers. No new readers will have references to the buckets beyond the common prefix.
6. Shrink the table's memory allocation.

To expand a hash table in place, we can make a similar adaptation to the expansion algorithm by adding a single wait-for-readers before setting the new size. However, the algorithm still requires auxiliary storage equal to the size of the current table. Together with the newly expanded allocation, this makes in-place expansion require the same amount of memory as out-of-place expansion.

4.3 Variation: Keeping Buckets Sorted

Typically, a hash table implementation will not enforce any ordering on the items within a hash bucket. This allows insertions to take constant time even if a bucket contains many items. However, if we keep the items in a bucket sorted carefully, modulo-based hashing will keep all the items destined for a given new bucket together in the same old bucket. This allows a resize increasing the size of the table to make only as many passes as the resize factor, minimizing the number of waits for readers. This approach optimizes resizes significantly.

Furthermore, an application may find sorted buckets useful for other reasons, such as optimizing failed lookups. Sorted buckets do not provide an algorithmic improvement for lookups, nor can they do anything to accelerate successful lookups; however, sorted buckets do allow failed lookups to terminate sooner, providing a constant-factor improvement for failed lookups and for removals. Blind inserts without checking for duplicates will incur a performance penalty to find the insertion point; however, insertions which check for duplicates will incur minimal additional cost.

Our hash-table expansion algorithm already performs a stable partition of the entries in a bucket, preserving the relative order of entries within each of the subsets that move to the buckets of the new table. The shrink algorithm, however, simply concatenates a set of old buckets into a single new bucket. A simple sort will not allow concatenation or splitting to preserve the sort, but a well-chosen sort order based on the hash key can allow concatenation without a merge step. Ori Shalev and Nir Shavit presented such a sorting mechanism in their "split-ordered list" proposal [22, 7]: they propose sorting by the bit-reversed key. Alternatively, the bucket selection could use the high-order bits of the hash key.

We do not pursue this variation further in this paper, but we do consider it a potentially productive avenue for future investigation.

5 Comparisons with Other Algorithms

We evaluated relativistic hash tables through both microbenchmarks on the data structure operations themselves, and via real-world benchmarks on an adapted version of the memcached key-value storage engine. The microbenchmarks directly compare our hash-table resize algorithm with two other resize algorithms: reader-writer locking and DDDS. The real-world benchmarks compare memcached's default storage engine with a modified memcached storage engine based on relativistic hash tables.

First, as a baseline, we implemented a simple resizable hash table based on reader-writer locking. In this implementation, lookups acquired a reader-writer lock for reading, to lock out concurrent resizes. Resizes acquired the reader-writer lock for writing, to lock out concurrent lookups. With lookups excluded, the resizer could simply allocate the new table, move all entries from the old table to the new, publish the new table, and reclaim the old table. We do not expect this implementation to scale well, but it represents the best-known method based on mutual exclusion, and we included it to provide a baseline for comparison.

For a more competitive comparison, we turned to Nick Piggin's "Dynamic Dynamic Data Structures" (DDDS) [21]. DDDS provides a generic algorithm to safely move nodes between any two data structures, given only the standard insertion, removal, and lookup operations for those structures. In particular, DDDS provides another method for resizing an RCU-protected hash table without outright blocking concurrent lookups (though it can delay them).

The DDDS algorithm uses two technologies to synchronize between resizes and lookups: RCU to detect when readers have finished with the old data structure, and a Linux construct called a *sequence counter* or *seqcount* to detect if a lookup races with a resize. A seqcount employs a counter incremented before and after moving each entry; the reader can use that counter, together with an appropriate read memory barrier, to check for a resize step running concurrently with any part of the read.

The DDDS lookup reader first checks for the presence of an old hash table, which indicates a concurrent resize. If present, the lookup proceeds via the concurrent-resize slow path; otherwise, the lookup uses a fast path that simply performs a lookup within the current hash table. The slow path uses a sequence counter to check for a race with a resize, then performs a lookup first in the current hash table and then in the old table. It returns the result of the first successful lookup, or loops if both lookups fail and the sequence counter indicates a race with a resize. Note that the potentially unbounded number of retries makes DDDS lookups non-wait-free, and could the-

oretically lead to a livelock, though in practice resizes do not occur frequently enough for a livelock to arise.

We expect DDDS to perform fairly competitively with relativistic hash tables. However, the DDDS lookup incurs more overhead than relativistic hash tables, due to the additional conditionals, the secondary table lookup, the expensive read memory barrier in the sequence counter, and the potential retries with a concurrent resize. Thus, we expect relativistic hash tables to outperform DDDS significantly when running a concurrent resize, and slightly even without a concurrent resize.

For a real-world benchmark, we chose memcached, a key-value storage engine widely used in Internet applications as a high-performance cache. Memcached stores key-value associations in a hash table, and supports a network protocol for setting and getting key-value pairs. Memcached also supports timed expiry of values, and eviction of values to limit maximum memory usage.

The default memcached storage engine makes extensive use of global locks. In particular, a single global lock guards all accesses to the hash table. As a result, we expect memcached's default engine to hit a hard scalability limit, beyond which it will not scale to more requests regardless of available resources.

Memcached requires the ability to scale to various workload sizes at runtime; as a result, it requires a resizable hash table. Previous non-resizable RCU hash tables could not provide the flexibility necessary for memcached.

We implemented a new RP-based storage engine in memcached, and modified memcached to support a new fast path for the GET request. memcached's default implementation goes to great lengths to avoid copying data when servicing a GET request; memcached also services multiple concurrent client connections per thread in an event-driven manner. As a result of these two constraints, memcached maintains reference counts on each key-value pair in the hash table, and holds a reference to the found item for a GET from the time of the hash lookup to the time the response gets written back to the client. In implementing the RP-based storage engine, we chose instead to copy the value out of a key-value pair while still within an RP reader; this allows the GET fast path to avoid interaction with the reference-counting mechanism entirely. The GET fast path checks the retrieved item for potential expiry or other conditions which would require mutating the store, and falls back to the slow path in those cases.

We expect that with the new RP-based storage engine, memcached will no longer hit the hard scalability limit observed with the default engine, and GET requests should continue to scale up to the limits of the test machine. Since we added wait-for-readers operations to the SET handling, SET will become marginally slower, but the scalability of SET requests should not change; we believe this tradeoff will prove acceptable in exchange for making GET requests scalable.

6 Benchmark Methodology

6.1 Microbenchmark: rcuhashbash-resize

To compare the performance and scalability of our algorithms to the alternatives, we created a test harness and benchmarking framework for resizable hash-table implementations. We chose to implement this framework as a Linux kernel module, `rcuhashbash-resize`. The Linux kernel already includes a scalable implementation of RCU, locking primitives, and linked list primitives. Furthermore, we created our hash-table resize algorithms with specific use cases of the Linux kernel in mind, such as the directory entry cache. This made the Linux kernel an ideal development and benchmarking environment.

The `rcuhashbash-resize` framework provides a common structure for hash tables based on Linux's hlist abstraction, a doubly-linked list with a single head pointer. On top of this common base, `rcuhashbash-resize` includes the lookup and resize functions for the three resizable hash-table implementations: our relativistic resizable hash table, DDDS, and the simple rwlock-based implementation.

The current Linux memory allocator supports shrinking memory allocations in place, but does not support growing in place. Thus, we implemented the in place variation of our shrink algorithm and the copying implementation of our expansion algorithm.

`rcuhashbash-resize` accepts the following configuration parameters:

- The name of the hash-table implementation to test.
- An initial and alternate hash table size, specified as a power of two.
- The number of entries to appear in the table.
- The number of reader threads to run.
- Whether to run a resize thread.

`rcuhashbash-resize` starts by creating a hash table with the specified number of buckets, and adds entries to it containing integer values from 0 to the specified upper bound. It then starts the reader threads and optional resize thread, which record statistics in thread-local variables to avoid the need for additional synchronization. When the test completes, `rcuhashbash-resize` stops all threads, sums their recorded statistics, and presents the results via the kernel message buffer.

The reader threads choose a random value from the range of values present in the table, look up that value,

and record a hit or miss. Since the readers only look up entries that should exist in the table, any miss would indicate a test failure.

The resize thread continuously resizes the hash table from the initial size to the alternate size and back. While continuous resizes do not necessarily reflect a common usage pattern for a hash table, they will most noticeably demonstrate the impact of resizes on concurrent lookups. In practice, most hash tables will choose growth factors and hysteresis to avoid frequent resizes, but such a workload would not allow accurate measurement of the impact of resizing on lookups. We consider a continuous resize a harsh benchmark, but one which a scalable concurrent implementation should handle reasonably. Furthermore, we can perform separate benchmark runs to evaluate the cost of the lookup in the absence of resizes.

The benchmark runs in this paper all used a hash table with 2^{16} entries. For each of the three implementations, we collected statistics for three cases: no resizing and 2^{13} buckets, no resizing and 2^{14} buckets, and continuous resizing between 2^{13} and 2^{14} buckets. We expect lookups to take less time in a table with more buckets, and thus if the resize algorithms have minimal impact on lookup performance, we would expect to see the number of lookups with a concurrent resizer fall between the no-resize cases with the smaller and larger tables.

For each set of test parameters, we performed 10 benchmark runs of 10 seconds each, and averaged the results.

Our test system had two Intel "Westmere" Xeon DP processors at 2.4GHz, each of which had 6 hardware cores of two logical threads each, for a total of 24 hardware-supported threads (henceforth referred to as "CPUs"). To observe scalability, we ran each benchmark with 1, 2, 4, 8, and 16 concurrent reader threads, with and without an additional resize thread. In all cases, we ran fewer threads than the hardware supported, thus minimizing the need to pass through the scheduler and allowing free CPUs to soak up any unremovable OS background noise. (We do however expect that performance may behave somewhat less than linearly when passing 12 threads, as that matches the number of hardware cores.)

All of our tests occurred on a Linux 2.6.37 kernel, targeting the x86-64 architecture. We used the default configuration (`make defconfig`), with the hierarchical RCU implementation, and no involuntary preemption.

6.2 Real-World Benchmarks: memcached

As a client-server program, memcached required a separate benchmarking program. At the recommendation of memcached developers, we used mc-benchmark, developed by Salvatore Sanfilippo. To minimize the impact of network overhead, we ran the client and server on the same system, communicating via the loopback interface. To generate enough load to reach the limits of memcached, the benchmarking program requires resources comparable to those supplied to memcached. Thus, on the same 24-CPU system, we chose to run 12 memcached threads and up to 12 benchmark processes.

mc-benchmark runs a single thread per process, but simulates multiple clients per process using the same kind of event-driven socket handling that memcached does. Experimentation showed that on the test system, one mc-benchmark process could run up to 4 simulated clients with increasing throughput, but at 4 clients it reached the limit of available CPU power, and adding additional clients would result in the same total request throughput. Thus, we ran from 1 to 12 mc-benchmark processes, each of which simulated 4 clients.

To run the memcached server and mc-benchmark client, and collect statistics on the request rate, we used a benchmark script supplied by the memcached developers. For each test run, the benchmark would start memcached and wait for it to initialize, start the desired number of concurrent mc-benchmark processes, wait 20 seconds for the processing to ramp up (mc-benchmark has to first run SET commands to insert test data, then either SET or GET requests depending on the benchmark), and then collect samples of the rate of processed requests directly from memcached; the benchmark collected three rate samples at 2 second intervals, and took the highest observed rate among those three samples.

7 Benchmark Results

To evaluate baseline reader performance in the absence of resizes, we first compare lookups per second for all the implementations with a fixed table size of 8192 buckets; figure 3 shows this comparison. As predicted, our relativistic hash table, shown as RP, and DDDS remain very competitive when not concurrently resizing, though as the number of concurrent readers increases, our implementation's performance pulls ahead of DDDS slightly. Reader-writer locking does not scale at all. In this test case, the reader-writer lock never gets acquired for writing, yet the overhead of the read lock acquisition prevents any reader parallelism.

We observe the expected deviation from linear growth for 16 readers, likely due to passing the limit of 12 hardware cores. In particular, notice that the performance for 16 threads appears approximately 50% more than that for 8, which agrees with the expected linear increase for fully utilizing 12 hardware cores rather than 8.

Figure 4 compares the lookups per second for our implementation and DDDS in the face of concurrent resizes. (We omit rwlock from this figure, because it would vanish against the horizontal axis; with 16 CPUs, rela-

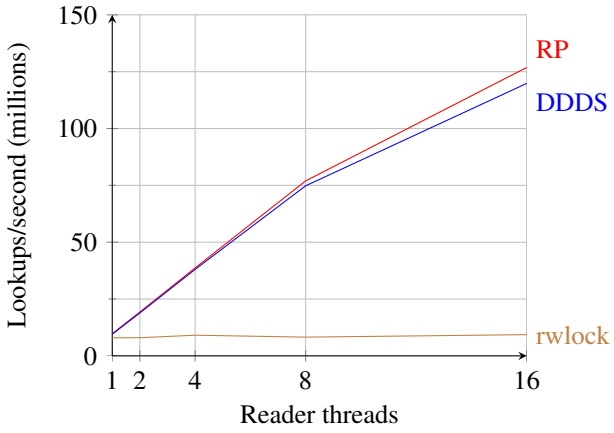


Figure 3: Lookups/second by number of reader threads for each of the three implementations, with a fixed hash-table size of 8k buckets, and no concurrent resizes.

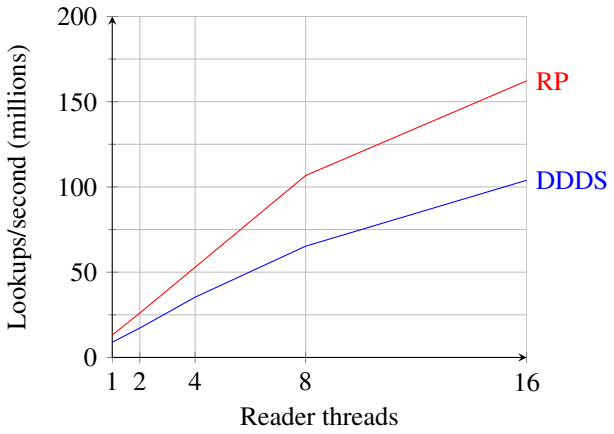


Figure 4: Lookups/second by number of reader threads for our RP-based implementation versus DDDS, with a concurrent resize thread continuously resizing the hash-table between 8k and 16k buckets. rwlock omitted as it vanishes against the horizontal axis.

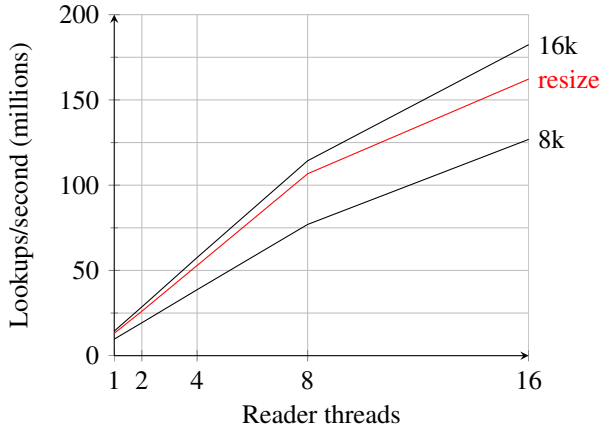


Figure 5: Lookups/second by number of reader threads for our resize algorithms. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

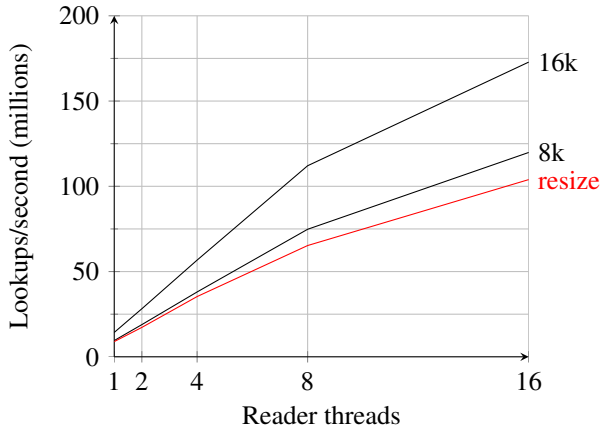


Figure 6: Lookups/second by number of reader threads for the DDDS resize algorithm. “8k” and “16k” indicate fixed hash-table sizes in buckets; “resize” indicates continuous resize between the two sizes.

tivistic hash tables provide 125 times the lookup rate of rwlock.) With a resizer running, our lookup rate scales better than DDDS, with its lead growing as the number of reader threads increases; with 16 threads, relativistic hashing provides 56% more lookups per second than DDDS. DDDS has sub-linear performance, while our lookup rate improves linearly with reader threads.

To more precisely evaluate the impact of resizing on lookup performance for each implementation, we compare the lookups per second when resizing to the no-resize cases for the larger and smaller table size. Figure 5 shows the results of this comparison for our implementation. The lookup rate with a concurrent resize falls between the no-resize runs for the two table sizes that the resizer toggles between. This suggests that our resize algorithms add little to no overhead to concurrent lookups.

Figure 6 shows the same comparison for the DDDS resize algorithm. In this case, the lookup rate with a resizer running falls below the lower bound of the smaller hash table. This suggests that the DDDS resizer adds significant overhead to concurrent lookups, as predicted.

Finally, figure 7 shows the same comparison for the rwlock-based implementation. With a resizer running, the rwlock-based lookups suffer greatly, falling initially by two orders of magnitude with a single reader, and struggling back up to only one order of magnitude down at the 16-reader mark.

Figure 8 shows the results of our benchmarks on mem-

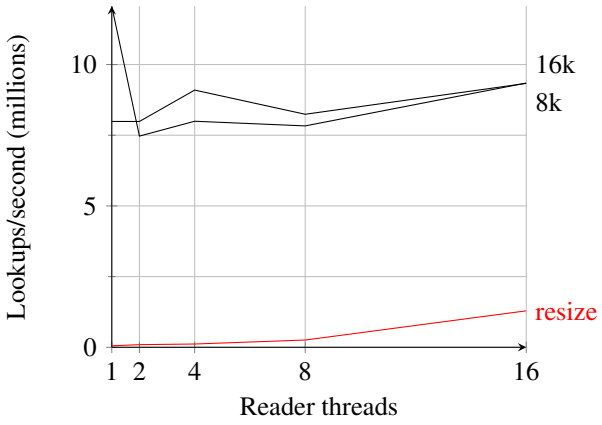


Figure 7: Lookups/second by number of reader threads for the rwlock-based implementation. "8k" and "16k" indicate fixed hash-table sizes in buckets; "resize" indicates continuous resize between the two sizes.

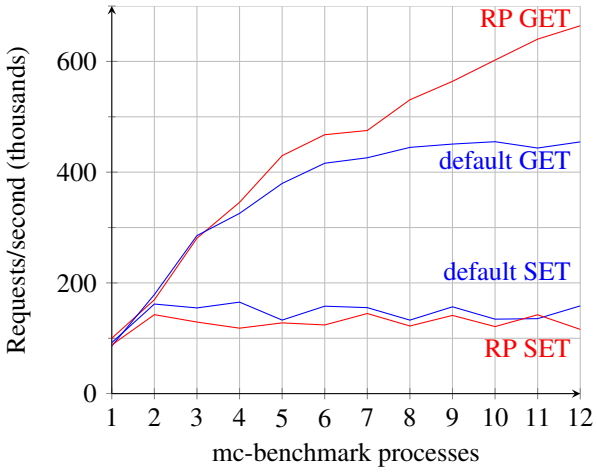


Figure 8: GET and SET operations per second by number of mc-benchmark processes for the default memcached storage engine and our RP-based storage engine. Each mc-benchmark process simulated 4 clients to saturate the CPU.

cached. Note that the default engine hits the expected hard limit on GET scalability, and fails to improve its request processing rate beyond that limit. The RP-based engine encounters no such scalability limit, and the GET rate grows steadily up to the limits of the system. With a full 12 client processes and 12 server threads, memcached with the RP-based engine services 46% more GET requests per second than the default engine.

As expected, SET requests do not scale in either engine. In the RP engine, SET requests incur the expected marginal performance hit due to wait-for-readers operations; however, this tradeoff will prove acceptable for many workloads, particularly when a successful GET request corresponds to a cache hit that can avoid a database query or other heavyweight processing.

We hypothesize that memcached's default engine only managed to scale to as many clients as it did because it spends the vast majority of its time in the kernel rather than in the memcached userspace code, and the kernel code supported more concurrency than the serialized engine code. Profiling confirmed that memcached spends several times as much time in the kernel as in userspace, regardless of storage engine.

We also performed separate runs of the benchmark using the mutex profiler mutrace. By doing so we observed that the default engine spent long periods of time contending for the global lock, whereas with the RP-based engine, GET requests no longer incurred any contention for the global lock.

7.1 Benchmark Summary

Our relativistic resizable hash table provides linearly scalable lookup performance in both microbenchmarks and real-world benchmarks. In our microbenchmarks, the relativistic implementation surpassed DDDS by a widening margin of up to 56% with 16 reader threads; both implementations vastly dwarfed reader-writer locks, with our RP implementation providing a 125x improvement with 16 readers. Furthermore, our resize algorithms minimized the impact of concurrent resizing on lookup performance, as demonstrated through the comparison with fixed-size hash tables. In the real-world benchmarks using memcached, the RP-based engine eliminated the hard scalability limit of the default storage engine, and consistently serviced more GET requests per second than the default engine—up to 46% more requests per second when saturating the machine with a full 12 client processes and 12 server threads.

8 Future Work

Our proposed hash-table resize algorithms demonstrate the use of the wait-for-reader operation to order update operations. We use this operation not merely as a write memory barrier, but as a means of flushing existing readers from a structure when their current position could otherwise cause them to see writes out of order. Figure 2 provided a specific example of this, in which a reader has already traversed past the location of an earlier write, but would subsequently encounter a later write if the writer did not first wait for such readers to finish.

We have developed a full methodology for ordering writes to any acyclic data structure while allowing concurrent readers, based on the order in which readers tra-

verse a data structure. This methodology allows writers to consider only the effect of any prefix of their writes, rather than any possible subset of those writes. This proves equivalent to allowing a reader to perform a full traversal of the data structure between any two write operations, but not overlapping any write operation. This methodology forms the foundation of our work on *relativistic programming*.

Relativistic readers traversing a data structure have a current position, or *read cursor*. Writes to a data structure also have a position relative to read cursors: some read cursors will subsequently pass through that write, while others have already passed that point. In an acyclic data structure, readers will start their read cursors at designated entry points, and advance their read cursors through the structure until they find what they needed to read or reach the end of their path.

When a writer performs two writes to the data structure, it needs to order those writes with respect to any potential read cursors that may observe them. These writes will either occur in the same direction as reader traversals (with the second write later than the first), or in the opposite direction (with the second write earlier than the first). If the second write occurs later, read cursors between the two writes may observe the second write and not the first; thus, the writer must wait for readers to finish before performing the second write. However, if the second write occurs earlier in the structure, no read cursor may observe the second write and subsequently fail to observe the first write in the same pass (if it reaches the location of the first); thus, the writer need only use the relativistic *publish* operation, which uses a simple write memory barrier.

"Laws of Order" [1] presents a set of constraints on concurrent algorithms, such that any algorithm meeting those constraints must necessarily use expensive synchronization instructions. In particular, these constraints include *strong non-commutativity*: multiple operations whose order affects the results of both. Our relativistic programming methodology allows readers to run without synchronization instructions, because at a minimum those readers do not execute strongly non-commutative operations: reordering a read and a write cannot affect the results of the write.

We originally developed a more complex hash-table resize operation, which required lookups to retry in a secondary hash table if the primary lookup failed; this approach mirrored that of the DDDS lookup slow path. Our work on the RP methodology motivated the simplified version that now appears in this paper. We plan to use the same methodology to develop algorithms for additional data structures not previously supported by RCU.

As an immediate example, the RP methodology allows a significantly simplified variation of our previous hash-table move algorithm [24, 23]. This variation will no longer need to copy the moved entry and remove the original, a limitation which breaks persistent references, and which made the original move algorithm unsuitable for use in the Linux dcache.

9 Availability

The authors have published the code supporting this paper as Free and Open Source Software under the GNU General Public License. For details, see `http://git.kernel.org/?p=linux/kernel/git/josh/rcuhashbash.git`.

10 Acknowledgments

Thanks to Nick Piggin, Linux kernel hacker and inventor of the DDDS algorithm, for reviewing our implementation of DDDS to ensure that it fairly represents his work. Thanks to Intel for access to the 24-way system used for benchmarking. Thanks to Jamey Sharp, Phil Howard, Eddie Kohler, and the USENIX ATC reviewers for their review, feedback, and advice. Thanks to memcached developer "dormando" for providing technical help with memcached, as well as the benchmarking framework. Thanks to Salvatore Sanfilippo for the original mc-benchmark.

Funding for this research provided by two Maseeh Graduate Fellowships, and by the National Science Foundation under Grant No. CNS-0719851. Thanks to Dr. Fariborz Maseeh and the National Science Foundation for their support.

References

[1] ATTIYA, H., GUERRAOUI, R., HENDLER, D., KUZNETSOV, P., MICHAEL, M. M., AND VECHEV, M. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated. In *Proceedings of the ACM POPL'11* (2011).

[2] CLICK, C. A Lock-Free Hash Table. In *JavaOne Conference* (2007).

[3] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, second ed. MIT Press, 2001, ch. Chapter 11: Hash Tables.

[4] DESNOYERS, M. *Low-Impact Operating System Tracing*. PhD thesis, École Polytechnique Montréal, 2009.

[5] GAO, H., GROOTE, J. F., AND HESSELINK, W. H. Lock-free dynamic hash tables with open addressing. *Distributed Computing 18*, 1 (July 2005).

[6] HART, T. E., MCKENNEY, P. E., BROWN, A. D., AND WALPOLE, J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing 67*, 12 (2007), 1270–1285.

[7] HERLIHY, M., AND SHAVIT, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008, ch. Chapter 13: Concurrent Hashing and Natural Parallelism.

[8] HOWARD, P. W., AND WALPOLE, J. A relativistic enhancement to software transactional memory. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 2011)* (2011).

[9] HOWARD, P. W., AND WALPOLE, J. Relativistic red-black trees. Tech. Rep. 1006, Portland State University, 2011. `http://www.cs.pdx.edu/pdfs/tr1006.pdf`.

[10] KNUTH, D. *The Art of Computer Programming*, second ed. Addison-Wesley, 1998, ch. Section 6.4: Hashing.

[11] LINDER, H., SARMA, D., AND SONI, M. Scalability of the directory entry cache. In *Ottawa Linux Symposium* (June 2002), pp. 289–300.

[12] MCKENNEY, P. E. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[13] MCKENNEY, P. E. RCU vs. locking performance on different CPUs. In *linux.conf.au* (Adelaide, Australia, January 2004).

[14] MCKENNEY, P. E. Software implementation of synchronous memory barriers. US Patent 6996812, February 2006.

[15] MCKENNEY, P. E. Sleepable read-copy update. Linux Weekly News. `http://lwn.net/Articles/202847/`, 2008.

[16] MCKENNEY, P. E., SARMA, D., ARCANGELI, A., KLEEN, A., KRIEGER, O., AND RUSSELL, R. Read Copy Update. In *Ottawa Linux Symposium* (June 2002), pp. 338–367.

[17] MCKENNEY, P. E., SARMA, D., AND SONI, M. Scaling dcache with RCU. *Linux Journal 2004*, 117 (2004).

[18] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems* (October 1998), pp. 509–518.

[19] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures* (2002), SPAA '02, pp. 73–82.

[20] MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (2004), 491–504.

[21] PIGGIN, N. ddds: "dynamic dynamic data structure" algorithm, for adaptive dcache hash table sizing. Linux kernel mailing list. `http://mid.gmane.org/20081007064834.GA5959@wotan.suse.de`, October 2008.

[22] SHALEV, O., AND SHAVIT, N. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM 53* (May 2006), 379–405.

[23] TRIPLETT, J. Lockless hash table lookups while performing key update on hash table element. US Patent 7668851, February 2010.

[24] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM Operating Systems Review 44*, 3 (July 2010).

[25] XU, H. bridge: Add core IGMP snooping support. Linux netdev mailing list. `http://mid.gmane.org/E1NlbuT-00021C-0b@gondolin.me.apana.org.au`, February 2010.

Evaluating the Effectiveness of Model-Based Power Characterization

John C. McCullough, Yuvraj Agarwal, Jaideep Chandrashekar[†]
Sathyanarayan Kuppuswamy, Alex C. Snoeren, and Rajesh K. Gupta

UC San Diego and [†]*Intel Labs, Berkeley*

Abstract

Accurate power characterization is important in computing platforms for several reasons ranging from power-aware adaptation to power provisioning. Power characterization is typically obtained through either direct measurements enabled by physical instrumentation or modeling based on hardware performance counters. We show, however, that linear-regression based modeling techniques commonly used in the literature work well only in restricted settings. These techniques frequently exhibit high prediction error in modern computing platforms due to inherent complexities such as multiple cores, hidden device states, and large dynamic power components.

Using a comprehensive measurement framework and an extensive set of benchmarks, we consider several more advanced modeling techniques and observe limited improvement. Our quantitative demonstration of the limitations of a variety of modeling techniques highlights the challenges posed by rising hardware complexity and variability and, thus, motivates the need for increased direct measurement of power consumption.

1 Introduction

Electrical power is a precious resource and its consumption is important to all forms of computing platforms from handheld devices to data centers. Numerous research efforts seek to optimize and carefully manage energy consumption at multiple levels—starting from individual components and subsystems such as wireless radios [27], storage devices [23], and processors [15, 28], to entire platforms [16, 39].

An important goal of these optimizations has been to achieve power proportionality, that is, power consumption that is proportional to the computational work done. As a result, a modern system's instantaneous power draw can vary dramatically. Moreover, the exact relationship between power draw and activity level is becoming increasingly complex due to the advent of microprocessors with multiple cores and built-in fine-grained thermal control, as well as hidden device states that are not necessarily exposed to the operating system.

While increasingly energy efficient components are a key enabler, achieving the goal of platform-wide power proportionality requires an intelligent dynamic power management (DPM) scheme. A critical first step towards that goal is to characterize how much power is being consumed, both by the platform as a whole and also by individual subsystems. Armed with a reasonable characterization of power consumption, a DPM system then needs to accurately predict how changes in utilization will impact future power consumption. For example, a DPM system can guide resource allocation decisions between heterogeneous but functionally similar resources such as multiple radios on the same platform [27].

Current approaches to prediction rely on assumptions of power proportionality to make architectural or system-level tradeoffs. These tradeoffs are evaluated by building a model of the system that describes power consumption in terms of power states and attributing the model to one or more observable hardware and software counters, correlating these with changes in measured power draw. Previous work on power modeling has focused on modeling total system power consumption using several learning techniques such as linear regression [21, 31], recursive learning automata [25], and stochastic power models [29]. The effectiveness of each of these techniques has been evaluated independently across various benchmarks (see [19] for a review).

However, we are unaware of any definitive comparison of these models for a diverse set of benchmarks on a given platform. Further, the increasingly nuanced relationship between a component's activity level and power consumption limits the utility of published models in modern systems. In particular, as we shall show, these models are effective only in a restricted set of cases, e.g., when system utilization is constant and very high, for

relatively straightforward executions in single cores, or when the system's static power consumption is dominant and the dynamic component is within the margin of error. Consequently, even well-designed DPM algorithms employing these models will make suboptimal decisions (to shutdown and/or slowdown components) if the actual utilization dynamically changes the significance of various components to overall power consumption.

To overcome the limited ability of models to predict power consumption, manufacturers often build reference systems complete with a large number of sense resistors and use precision analog-to-digital converters (ADC) to create a measurement instrument that accurately captures power consumption at fine time granularities. Research efforts have mimicked this approach by developing custom designs that can breakdown power consumption within sensor platforms [35] or monitor whole-system power for general purpose computers at the power supply [10]. While such extensive direct instrumentation can provide accurate power measurements, it requires significant design effort and increases costs due to board space constraints and the need for additional components. Hence, we are not aware of any production systems so instrumented. Moreover, significant increases in cross-part variability [13, 38] limit the applicability of predictions based upon even the measured behavior of a single or small number of reference systems.

We evaluate the need for pervasive power instrumentation by exploring the effectiveness of power modeling on modern hardware. Mindful of the fact that the required level of accuracy varies based upon the specific DPM goal, we consider how well increasingly sophisticated models can predict the power consumption of realistic workloads. We make the following contributions:

- We show empirically that while total system power can be modeled with 1–3% mean relative error across workloads when restricted to a single core, it rises to 2-6% for multi-core benchmarks.

- We find that linear models have significantly higher mean relative error for individual subsystems such as the CPU: 10–14% error on average, but as high as 150% for some workloads. We employ more complex techniques to improve predictive performance, but only by a few percent.

- We present an in-depth analysis of why modeling fails for modern platforms, especially under multi-core workloads. We posit that this poor predictive performance is due to effects such as cache contention, processor performance optimizations, and hidden device states not exposed to the OS. In addition, we present quantitative evidence of significant variability between identical components, which fundamentally bounds the potential accuracy of any modeling based approach.

Taken together these results make a strong case for pervasive instrumentation and dynamic collection of power usage data. While traditionally eschewed due to significant increases in design and manufacturing costs, the advent of relatively inexpensive ADCs and associated circuits makes such an approach increasingly feasible.

2 Related work

Researchers have long been interested in optimizing the energy efficiency of computing devices. In the context of mobile platforms, researchers have considered optimizing individual subsystems such as the CPU [15, 28], disk [23] and wireless radios [27]. While much of the early work focused on battery powered devices for usability reasons [16, 27, 39], economic motivations have dramatically increased interest in general purpose computing such as PCs and servers [1, 3, 26].

There have been a number of efforts to predict energy consumption through in-situ power measurements by adding different levels of hardware instrumentation. The Openmoko Freerunner mobile phone platform was designed to support developer access. Carroll and Heiser leveraged its sense-resistor support to characterize power consumption at a component level, deriving simple, time-based linear models for each component [7]. The LEAP platform [35] proposes adding fine-grained instrumentation to all power rails in embedded sensor nodes, and develops the required software instrumentation within the OS kernel to attribute energy to running tasks. In contrast, Quanto [17] proposes a single point of measurement by observing the switching regulator on sensor platforms to increase the practicality of energy attribution. Quanto requires changes to TinyOS and individual applications to track individual state transitions and offline analysis for energy attribution.

These measurement efforts, however, are restricted to special-purpose platforms with limited availability. Moreover, while such detailed, instrumentation-based approaches provide accurate power characterization, they are frequently regarded as too costly to implement at scale. Hence, efforts focused on more general purpose platforms have typically relied upon modeling, mostly for predicting total system power.

Recent activity in server environments is driven by the observation that most computer systems are largely idle [1] or exhibit low utilization except during peak traffic [3]. Hence, Barroso and Hölze argue for more energy-proportional designs so that the energy used is propor-

tional to the utilization [3]. This quest for energy proportionality has led to a variety of low-power server designs [2]. Some applications, however, have been shown to perform poorly on these low-power platforms [30].

Predicting the energy use of a particular application on a general-purpose platform is challenging, however. Previous studies have shown that CPU utilization is not useful by itself and that performance counters used to build models only help if they are correlated with dynamic power [24, 31]. Economou et al. explored the use of component-level measurements including CPU, memory and hard disk but were unable to significantly decrease prediction error [14]. Conversely, Bircher and John explored using performance counters to model not only total system power, but component-level power (e.g., CPU, memory, disk, and I/O) as well in a Pentium IV-class system [5]. They find that linear models are a poor fit for all but the CPU, and instead resort to multiple-input quadratics to provide an average of 9% absolute error.

In the hosted setting, Koller et al. turn to application-level throughput—as opposed to hardware performance counters—to improve power prediction for virtual clusters using linear combinations [22]. The Joulemeter project [21] proposes combining total-system power measurements from server power supplies with power modeling to attribute power consumption at the coarse level of individual virtual machines (VM) running on a physical server. Their model is generated by exercising specific VMs by running particular applications and using the CPU utilization metric to attribute energy.

Despite prediction errors, a number of researchers have demonstrated the utility of power modeling. For example, Bircher and John combine CPU metrics with instantaneous voltage demand information to ensure that the processor uses the correct DVFS setting and achieve an average speedup of 7.3% over Windows Vista's default DVFS algorithm [6]. Wang and Wang utilize feedback control theory to jointly optimize application-level performance and power consumption [37], while Tolia et al. improve power proportionality by leveraging virtual machines, DVFS, and fan control [36].

3 Power characterization

Characterizing the power consumption of a computing platform need not be difficult in principle. Ideally, original equipment manufacturers (OEMs) are well positioned to add extensive power instrumentation to their platforms, which would enable accurate and fine grained power measurements. Combined with such instrumentation, OEMs could further expose interfaces to an operating system to query detailed power information in a low-overhead manner. This information can then be used by the OS as well as individual applications to manage their energy consumption dynamically. Unfortunately, this ideal scenario is not realized in practice due to manufacturing constraints such as increased board area, cost of components and design costs. Modern platforms are already extremely complex and OEMs are reluctant to add functionality without clear and quantifiable benefits. Hence, while OEMs may have extensive power instrumentation on their *development* platforms during design and testing, we are unaware of any commodity platform that provides fine-grained power measurement capabilities in hardware.

Instead, in the absence of direct power measurement, the commonly used alternative is to make power models. The basic idea behind power modeling is to take as input various software and hardware counters and use those to predict power consumption after suitable training on an appropriately instrumented platform. Regardless, any power characterization approach, whether based upon modeling or direct instrumentation, must trade off between several design alternatives as discussed below.

3.1 Measurement granularity

One of the dimensions that affects both forms of power characterization—power measurement and power modeling—is the granularity of measurement. Power can be characterized at the level of an entire system (a single power value) or can be done at a logical subsystem granularity, such as the display, CPU, memory and storage subsystems. The appropriate measurement granularity depends on the application. For example, in data centers an application for macro-scale workload consolidation on servers will likely only require total system power measurements at an individual server level. On the other hand, fine-grained scheduling decisions on individual heterogeneous processor cores requires power consumption data for individual cores.

While total system power can be measured at the wall socket directly using myriad commercial devices (e.g. `WattsUP` meters) the applicability is limited in the case of any fine grained adaptation. Furthermore, power measurements at the system level cannot distinguish between the actual power used and the power wasted due to the inefficiencies in the power supply. On the other hand, fine-grained subsystem level power characterization is more useful since the total system power can still be estimated accurately by adding the power consumption of individual components. Most of the research to date has focused on total system power modeling [14, 31]. In this paper we explore the design space of power characterization and especially investigate the feasibility of accurate power modeling at subsystem granularity.

3.2 Accuracy

In the case of power instrumentation, it is possible to get very high levels of accuracy with precision analog-to-digital converters (ADCs), albeit at higher costs. The accuracy of power modeling with respect to the ground-truth measurements is important since the accuracy depends on how well the model fits. Similar to the measurement granularity dimension, the accuracy requirements of power modeling are also somewhat dependent on the application. When consolidating workload to fewer servers in a data center, modeling total power within 5–10% error is sufficient to guide policy decisions since the base power of servers is high [14, 31]. On the other hand, for an application of fine grained scheduling on different heterogeneous processor cores, the accuracy of power characterization needs to be at least as good as the differences in power consumption between the cores, otherwise scheduling decisions may be incorrect.

Furthermore, the required accuracy is likely to vary with particular subsystems based on factors such as their dynamic range, and their contribution to the total system power. Subsystems that are dominant in particular platforms need to be measured more accurately since the penalty of mis-predicting the power is higher. Furthermore, subsystems that are more complex and dynamic, such as processors, need higher accuracy measurements. On our test systems, the CPU consumes between 0.5W and 27W (constituting up to 40% of the total system power), and prediction errors translate to high absolute error. In contrast, an SSD disk drive on the SATA interface, or the network interface, consumes lower power (fewer than 2–3W) and is less dynamic; therefore, higher modeling errors can be tolerated. However, in cases of platforms with a larger number of disks this modeling error will have a more significant impact.

3.3 Overhead and complexity

Both power modeling and power instrumentation have associated overheads and complexity. In the case of power instrumentation, OEMs have to integrate the power measurement functionality into their platforms, usually in the form of a shunt resistor connected to ADCs. The ADCs measure the voltage drop across the shunts, which is converted into current and power consumption. Since modern platforms have multiple voltage domains, and subsystems can be powered using +3.3V, +5V, and +12V power supplies, a large number of ADC inputs are required. Furthermore, in case the voltage to the subsystem is not constant, such as with processors that employ dynamic voltage scaling, we need additional ADC inputs to measure voltage as well. Some subsystems, such as processors, can also have multiple power lines powering different functional units within them which each need to be measured separately. The shunt resistors themselves need to be chosen while keeping in mind the dynamic range of the power consumed by individual subsystems, possibly from several milliwatts to tens of watts, to give high precision and low power loss in the sense resistor itself. All of these factors contribute to higher costs—of components, board area and design, test and validation time.

Complexity in power modeling arises in part from the need to capture all the relevant features expressed by software and hardware counters to serve as inputs to build the models. Often platforms and components, such as the CPU, either support tracking a limited set of platform counters simultaneously, or have-non trivial overhead in collecting a large set of counters at fine granularities. Therefore, it is important that the model be sparse and use as few counters and states are possible, while still providing reasonable modeling accuracy. Additionally, the models themselves can be arbitrarily complex and can require non-trivial amounts of computation. In this paper, we explore a series of increasingly complex models to understand the accuracy/complexity tradeoffs.

Finally, in the case of power modeling, transferability or robustness of the power modeling is key: The model should be generated once and should be applicable over time and to other instances of the same platform. While we believe that this is intuitively the case, recent work has highlighted a significant amount of platform heterogeneity, where process and manufacturing variations and aging effects lead to identical hardware exhibiting significant variation in power consumption [8, 13, 38]. We show some preliminary results relating to this aspect and highlight the associated challenges with power modeling in the face of increasing variability in hardware.

4 Power modeling

While there has been a considerable volume of work in the area of power modeling, notably for system power and CPU power, a common toeu hread joining most of the previous work is the assumption that system power can be well predicted by simple linear regression models [14, 21, 31]. Our goal in this paper is to understand whether (i) these simple models are compatible with more contemporary platforms (CPU and platform complexity has increased significantly since many of the previous modeling approaches were proposed), and (ii) whether these models can be applied to individual subsystems within platforms. The latter is important to understand because, with the increasing emphasis on power proportionality and energy awareness, there are several adaptations that can be done at the platform level as well

as the subsystem level, provided fine-grained power consumption information is available.

Our initial attempts to use simple linear regression models—including replicating specific ones previously proposed—were disappointing: The models perform poorly on non-trivial workloads. This result could be explained by one of the following reasons:

- The features being fed into the model contain a certain level of cross-dependency, whereas linear models assume feature independence.
- The features used by previously published models are no longer appropriate for contemporary platforms. There may, however, exist a different set of counters that can still lead to a good model.
- Modern hardware components, such as processors, abstract away hardware complexity and do not necessarily expose all the power states to the OS and are thus fundamentally hard to model since changes in power consumption are not necessarily associated with changes in exposed states.

In this section, we describe a number of increasingly complex regression models that we use to fit the power data. Unfortunately, we found that increasing the complexity of the model does not always improve the accuracy of power prediction across the different subsystems that we are trying to model. We begin by describing how linear regression models are constructed and enumerating the specific models we use.

4.1 Linear regression models

Let $\mathbf{y} = [y_1, y_2, \ldots, y_k]$ be the vector of power measurements and let $\mathbf{x}_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,n}]$ be the normalized vector of measurements on the n variables (hardware counters and OS variables) collected at time t_i. The linear regression model is expressed as:

$$y_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \ldots + \beta_n x_{i,n} + \epsilon_i,$$

where ϵ_i is a noise term that can account for measurement error. Thus, the linear regression models are solved by estimating the model parameters β, and this is typically done by finding the least squares solution to $\hat{\beta} = \mathbf{y}.\mathbf{X}^{-1}$, which can be computed as

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^{k} \left(y_i - \beta_0 - \sum_{j=1}^{n} \beta_j x_{i,j} \right)^2,$$

or simply $\hat{\beta} = \operatorname{argmin}_\beta \left(||\mathbf{y} - \beta \mathbf{X}||_2^2 \right)$.

The challenge in building a good power model is to correctly identify the set of n most relevant features. On the platforms we considered, there are in excess of 800 different hardware counters that can be tracked (even though only a few can be tracked *simultaneously*). Previous work has overcome this problem by using domain knowledge about the platform architecture to hand pick counters that are believed to be relevant [14, 31]. We believe that increasing complexity in modern processors and platforms makes this task harder with each generation. To understand whether such domain knowledge is critical to power modeling, we also use modeling techniques that perform automatic feature selection in the process of constructing a model. We observe that the features selected by the more complex techniques correspond to the features with the highest mutual information [11] for a given power rail. This makes us confident that these state-of-the-art modeling techniques are leveraging all relevant features and are not missing anything that is relevant but not linearly correlated.

We now briefly describe the regression techniques that we explore, listed in order of increasing complexity.

MANTIS: Ecomomou et al. developed a server power model by fitting a linear regression of four distinct utilization counters obtained from different platform subsystems to the power measurements taken at a wall socket [14]. The input utilization metrics are obtained by running a number of systematic workloads that stress the platform subsystems in sequence.

In particular, they consider counters corresponding to CPU utilization, off-chip memory accesses, and hard drive and network I/O rates. We extend the basic MANTIS linear model to also consider instructions per cycle as a representative baseline obtained from a best-in-class full-system power model [31]. While there have been a large number of efforts focused on identifying a suitable set of performance counters, we choose the MANTIS model as a starting point because it has been shown to have very good predictive properties on previous-generation hardware [14]. Since a few of the counters used by the original MANTIS model are no longer available on modern platforms, we communicated with the authors themselves to find appropriate substitutes.

Lasso regression: There are two drawbacks to building a (linear) regression model. First, it requires domain knowledge to identify the correct set of features (possibly from a large space)—in this case, we seek features possibly related to power consumption. Second, when the features are correlated to each other, the regression simply distributes the coefficient weights across the correlated features, and all correlated features are included in the final model, rather than identifying a smaller subset of somewhat independent features. In current complex platforms, there is a very large space of features that can be measured and it is a non-trivial task—even

for an expert—to correctly identify the smallest possible subset of power relevant features [33]. Lasso regression, which is a specific instance of l_1-regularized regression, overcomes this challenge by penalizing the use of too many features. Thus, it tends to favor the construction of sparse models which incorporate just enough features as are necessary. This is done by incorporating a penalty factor into the least-squares solution for the regression, which is expressed as:

$$\hat{\beta} = \underset{\beta}{\operatorname{argmin}} \left(\sum_{i=1}^{k} \left(y_i - \beta_0 - \sum_{j=1}^{n} \beta_j x_{i,j} \right)^2 + \lambda \sum_{j=1}^{d} |\beta_j| \right)$$

or, simply $\hat{\beta} = \operatorname{argmin}_{\beta} \left(||\mathbf{y} - \beta \mathbf{X}||_2^2 + \lambda \, ||\beta||_2 \right)$.

Here, λ is a penalty function that encourages the solution to aggressively set β values to zero (and exclude the associated features from the model). Compared to regular methods, Lasso regression is advantageous since it relies less on strong domain knowledge to pick out the right features; in addition, it is computationally simple, and automatically picks models with a small number of features, which are critical requirements for a usable power model. The optimal value for the λ parameter is selected by cross validation on the training data. We used the `glmnet` package to perform the Lasso regression [18].

4.2 Non-linear regression models

Linear regression models work well when the features being modeled are independent of each other and tend to predict poorly when there are interdependencies between the modeled features; non-linear models can often capture these feature dependencies. (Indeed, previous work has shown that quadratic models can be more effective at modeling subsystem power [5].) The non-linear form of the model can be expressed as:

$$y_i = \beta_0 + \sum_{\ell=1}^{m} \beta_\ell \phi_\ell(x_i) + \epsilon_i,$$

where ϕ_ℓ are non-linear basis functions over the feature vectors $\mathbf{x}_i$. We use the Lasso least-squares formulation as before to solve the regression and construct a model.

In general, the set of possible ϕ_j is arbitrarily large and solutions exist for only a few families. We experiment with three well known functions:

Polynomial with Lasso: Here, the basis functions are defined as exponentiated forms of the original variables. So, $\phi = \{x_i^a : 1 \leq a \leq d\}$ where $d = 3$. Again, with Lasso, only the relevant features—now including the polynomial terms which may have cross dependencies—are inserted into the model.

Polynomial + exponential with Lasso: In this slight variation of the previous model, ϕ also includes the functions e^{x_i}. As before, we run the full set of terms through the Lasso (linear) regression package which picks out a sparse subset of the terms. In the previous case as well as this one, the optimal λ is selected by cross validation.

Support vector regression (SVR): We also experiment with support vector machine (SVM)-based regression. At a high level, SVMs operate by fitting a hyperplane decision boundary to a set of labeled data instances taken from different classes. The hyperplane boundary is constructed so as to maximize the separation between the two classes of data, and can be used to perform classification, regression, and function estimation tasks on the data. SVMs employ a trick to handle non-linearity, the data is run through a non-linear kernel that maps the data to a higher dimensional space where greater separation may be achieved. An important difference between SVR and Lasso-based methods is that SVR does not force the regression to be sparse. When features are correlated, weights are distributed across them. We use the `libsvm` [9] and, in particular, the radial basis kernel. The parameters required for the RBF kernel were optimally selected by cross validation on the training data.

5 Evaluation setup

We collect platform subsystem power using an instrumented Intel Calpella platform. The platform is a customer reference board that corresponds to the commercially available mobile Calpella platform. This particular board, which is based on the Nehalem processor architecture, was outfitted with an Intel quad-core i7-820QM processor, 2x2GB of DDR3-1033 memory and a SATA SSD drive, running the 2.6.37 Linux kernel. Importantly, we turned off HyperThreading and TurboBoost on the platform to avoid hidden states (these states change operating points but are controlled in hardware and the OS has little visibility into them). A salient feature of this particular board is that it has been extensively instrumented with a very large number of low-tolerance power sense resistors that support direct and accurate power measurements of various subsystems (by connecting to a data acquisition system). The platform contains over one hundred sense resistors and it is a non-trivial task to collect readings from all of them. Instead, we first identified the platform subsystems that were of interest to us and simply instrumented the resistors for those subsystems and connected them to two 32-input National Instrument USB6218 DAQs. Finally to measure the total power consumption at the wall we use a commercial WattsUp meter. The Calpella platform is powered soley by a 12-V input from the ATX power connector and we consider

Subsystem	# resistors	min–max
CPU core	3	0.5–27W
CPU uncore (L3, mem. controller)	1	1–9W
integrated graphics	2	n.a.
discrete graphics	2	≈15.3W
memory	2	1–5W
CPU fan	1	≈0.7W
SATA	3	1.3–3.6W
LAN	1	≈ 0.95W
Chipset + other	0	0.5–5W
12V ATX in	1	23–67W

Table 1: Power characterization for the calpella platform. The 500-W ATX PSU that we use dissipates 20–26W due to conversion inefficiency and is not shown.

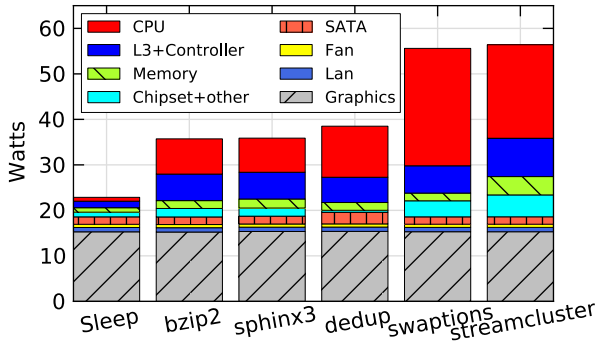


Figure 1: Power breakdown for sample workloads.

the 12-V rail to represent the total system power in order to eliminate the influence of variable power-supply inefficiencies. The power breakdown of the various subsystems is shown in Table 1.

The 16-bit NI-DAQs have a worst case error of 0.02%, but to scale the measured voltages to the range of the NI-DAQ we use precision voltage dividers, which in turn introduce a 0.035% measurement error, leading to an overall error of 0.04%. To minimize measurement overhead, the data from the NI-DAQs and the WattsUp meter is collected on a separate computer.

Along with the (externally) collected power readings, we collect OS-level statistics, hardware states and performance counters from the platform itself (for simplicity we will use the terms *counter* and *state* interchangeably). We extract OS-level statistics from `/proc` and `/sys` in Linux; these include processor utilization, disk usage, and processor C-state residency statistics. We collect hardware performance counters using the Linux `perf_event` framework.

By default the `perf_event` framework provides access to the four programmable and the six fixed hardware counters available per core. As a departure from previous processor models, Nehalem processors introduce "uncore" counters, which measure the performance of the L3 cache, QPI bus, and the memory controller. To replicate the MANTIS model, we need to measure last-level cache misses in the L3 cache. Fortunately, we have a kernel patch that provides access to the eight programmable per-socket performance counters. The measurement framework reports a total of 884 counters. While it would be ideal to measure them all concurrently and allow the models to pick out the most relevant features, the small number of programmable counters that can be read concurrently makes this task impossible.

Instead, we use a simple heuristic to reduce this number to a more manageable size: we sweep through the entire set of possible counters, making sure to get at least one run for each counter; then we compute the correlation of each counter with the total system power and discard all the counters that show no variation (with power), or that have very poor correlation. This brings down the set of potential counters to about 200, which is still large. To bridge the gap, we select all of the OS counters (these can be measured concurrently), and we greedily add as many hardware counters, in order of their correlation coefficients, as we can measure concurrently.

Note that due to issues with the aging OS required for NI-DAQ driver support, our test harness is initiated from an external machine. A high-level diagram of the measurement setup is shown in Figure 2. We have set up the NI-DAQ to sample each ADC channel at 10Khz and output average power consumption for each subsystem once per second for accurate power measurements. In our setup, we thus collect power readings as well as the on-platform measurements at a one-second granularity. Adjusting the collection granularity does not appreciably impact the prediction accuracy, and we feel that one second is a reasonable compromise: sampling the OS level counters at a faster rate would incur a higher overhead and introduce stronger measurement artifacts (where the act of measuring itself takes a non-trivial amount of power), while sampling it any slower might limit how quickly applications can react to changes in power consumption.

5.1 Benchmarks

To systematically exercise all possible states of the platform, particularly the subsystems that we are measuring, we selected an array of benchmarks from two well known benchmarking suites, as well as a few additional benchmarks to extend the subsystem coverage. We include the majority of the benchmarks in the SpecCPU benchmark suite [34]. We include 22 of the 32 benchmarks, excluding ten because they would either not com-

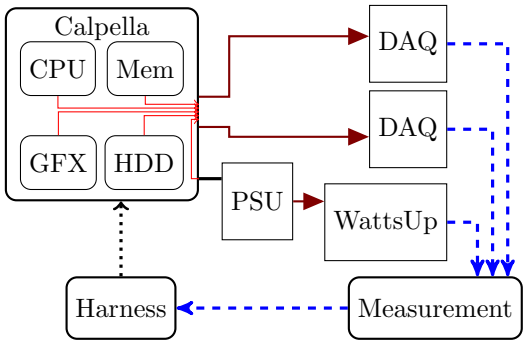


Figure 2: Test harness configuration.

pile with a modern compiler or tended to exhaust memory. While the SpecCPU suite is a well established benchmark, it only contains single-threaded benchmarks. To get a more representative set of benchmarks that would better exercise our multi-core system, we included the PARSEC [4] benchmark suite; this consists of a range of multi threaded "emerging workloads", including file de-duplication and x264 compression. We also include the `Bonnie` I/O benchmark, a parallel `LinuxBuild` kernel compile, `StressAppTest` [32], a `memcached` workload, as well as a synthetic `cpuload` benchmark. The StressAppTest program is a burn-in program designed to place a realistic high load on a system to test the hardware devices. We observe that while most of these benchmarks use the hardware as quickly as possible, evidence suggests that systems are not always fully loaded [3]. To capture this behavior, we supply `memcached` with a variety of different request rates to target different utilizations, and we duty cycle our synthetic floating point `cpuload` benchmark. Finally, we include `Sleep` to represent the system at idle.

5.2 Modeling evaluation

The effectiveness of a model is often decided by learning the model from a training set of data, and then assessing its predictive performance on a (different) testing set. Selecting the training set is often a non-trivial task and must ensure that the training set includes enough samples from various operating points. When this is not done judiciously the testing error can be large, even though the training error is small. The ideal scenario is to identify a set of "basis" benchmarks that are known to provide the sufficient coverage and to generate the training data from these benchmarks (a form of this was done in [31]). However, this is hard to achieve when systems are complex and have a large operating space. When we tried such an approach, the results were disappointing and led us to ask a more basic question: how well does the model work when the testing and training data are similar? This puts the focus on whether good models can be generated at all, rather than picking the smallest set of workloads needed to construct a good model. We employ a well known validation technique known as $k \times 2$ cross-validation. For this technique, we randomize the ordering of the data (collected from all the benchmarks), partition into two halves, use the first half as training data and learn the model, and then compute the prediction error on the latter half. The process is repeated multiple times (we repeat 10 times) and the errors from each run are aggregated.

We note that in our experimental evaluation, the error observed on the testing data set is approximately the same as that on the training data set, not only when the error is low, but also when the error is high. This raises our confidence that the models obtained each time are sufficiently general. In the next section, we present results from building models on specific platform subsystems.

6 Results

In this section we present results from evaluating the various models on several different operating configurations. The metric we use to test the efficacy of a model is *mean relative error*, which we shorten to *error* in the discussion, defined as follows:

$$error = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\hat{p}_i - p_i}{p_i}\right|$$

where $\hat{p}_i$ is the model predicted power, and p_i is the actual power measurement. For the results shown in this section, we execute each benchmark and configuration five times ($n = 20$). We note that the metric used is consistent with previous work [31]. One point of departure with previous work is that we measure "system power" after the PSU, and hence we capture a more accurate reflection of the actual power being consumed.

After running a large number of experiments across a variety of configurations, we can reduce the findings into three takeaways, which we discuss next.

6.1 Single core

To reduce number of variables we first limit the system to use one processor core only and run all the benchmarks on that single core. HyperThreading and TurboBoost were also turned off, and the processor P-state (frequency) was fixed. This is a reasonable approximation to the systems that were used to develop the MANTIS model. For this configuration, we note the following: the mean prediction error (averaged over all the benchmarks to obtain a single number) for total system power is between 1–3%. The errors are also low for platform subsystems: for example the average error in CPU power

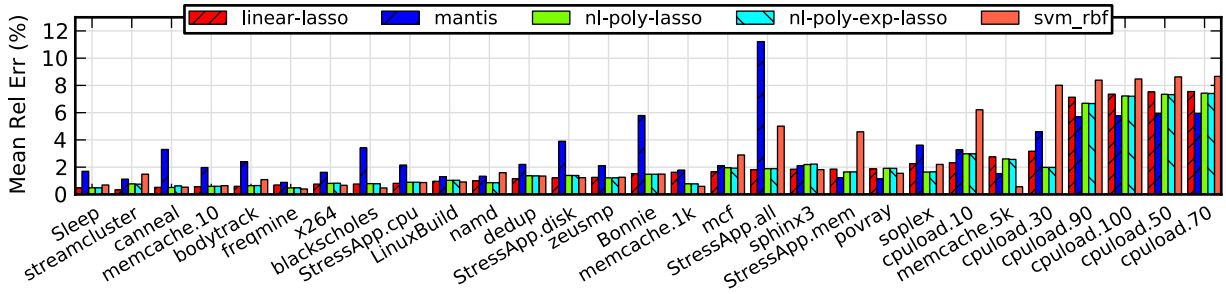


Figure 3: Modeling acuracy for total system power (Single-Core). Mean relative error is 1–3% across workloads.

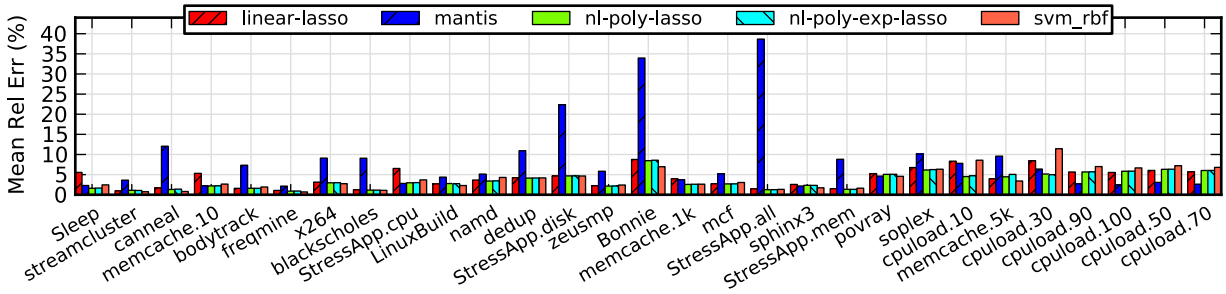


Figure 4: Modeling acuracy for CPU power (Single-Core). Mean relative error is 2-6% across workloads.

prediction is 2–6%, and the average error in predicting memory power is about 4–8% across the different models. Due to space constraints we do not show the results for the other subsystems, however the prediction errors is similar to that of the CPU.

In this context, mean prediction error can be misleading because the mean is computed over all time samples and is weighted towards longer-running benchmarks. Thus, the error will be worse if the system is executing a specific workload that is poorly predicted.

Figure 3 compares the prediction error in total system power over all the benchmarks used. The models do rather well, the majority of benchmarks show an error of less than 5%. The MANTIS model does well (which we expected), but the linear-Lasso model does slightly better than do the non-linear Lasso models. Upon closer inspection we observe that this ordering of results is tied to how well the models predict the CPU power, which along with a considerable base power factor, is a large contributor to the total system power.

Figure 4 compares the prediction errors across different models for the CPU subsystem power. Here, the differences between the models are more pronounced. We see that the MANTIS model is off by at most a few percent for most of the benchmarks, except for `canneal`, `StressApp`, and `bonnie`, which have high utilization, low IPC, and consequently lower power than the model attributes to utilization alone. It is important to note that the linear-Lasso model, which picks out the set of features automatically, consistently outperforms the MANTIS model, which uses domain knowledge to select the features. Not surprisingly, the set of counters picked out by linear-Lasso is a superset of the counters used by the MANTIS model; the C-state counters included in the linear-Lasso model, but not the MANTIS model seem to improve predictive power. Thus, this goes to establish that as systems become increasingly complex, the task of applying domain knowledge to pick out the most accurate set of counters becomes progressively harder and techniques that do automatic feature selection will be very useful in building effective models.

Finally, Figure 5 compares the error in prediction in the memory subsystem for different models and across various benchmarks. Similar to the CPU, all models save for SVM-rbf, do quite well and have comparable errors. Also, when compared to CPU power, the prediction error for different models are similar. This hints at the fact that all models use the same set of relevant features, which for the memory subsystem (which is simple) is quite predictable—L3-cache-misses being the most relevant and dominant feature.

Most of the results discussed so far were as expected—linear models have been shown to work well to predict full system power [14, 31]. Promisingly, even with an increase in subsystem power management complexity over prior work [5], the same linear models also do well in predicting platform subsystems.

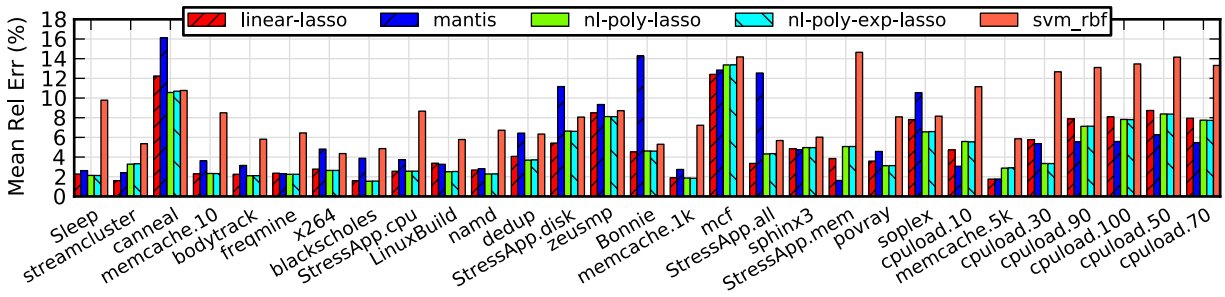


Figure 5: Modeling acuracy for memory power (Single-Core). Mean relative error is 4–8% across workloads.

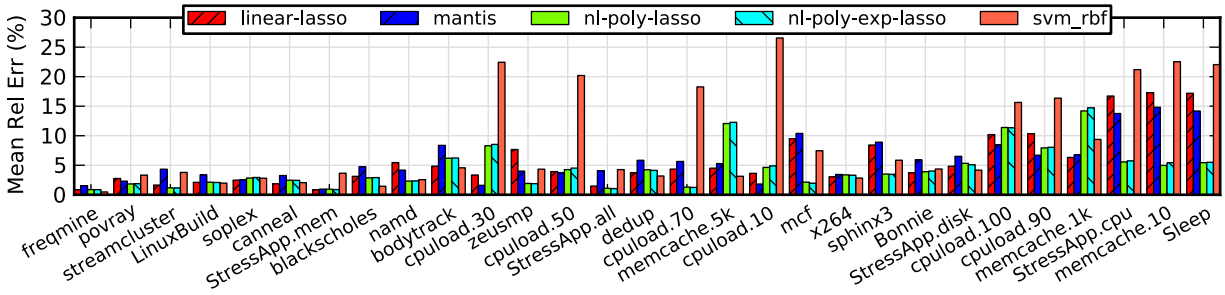


Figure 6: Modeling acuracy for total system power (Multi-core). Mean relative error is 2–6% across workloads.

6.2 Multicore

Next, we move to a more complex, yet more representative system configuration that utilizes all four available cores (HyperThreading and TurboBoost are still disabled and the P-state is still fixed). While more realistic, it still insulates the system from the extra hidden states of DVFS and functional contention, making power as easy to predict as possible. Figure 6 shows that across all the benchmarks, total system power is predicted to within 2-6%, which is respectable. This is well within the accuracy range required for tasks like data center server consolidation. However, as shown in Figure 7, the prediction error for CPU power is significantly higher, compared to that of the single core configuration, at 10-14%.

If we look more closely at the individual benchmarks for system power (Figure 6), we see that the error varies drastically by particular benchmark. Some benchmarks are predicted quite well (those near the left of the bar graph) and others do rather poorly (those to the right). Interestingly, the ordering of models changes with each benchmark, i.e., a particular model does not consistently do better than another over the entire set of benchmarks. In every benchmark there is at least one model that has an error less than 6%, but it is not always the same model. Our intuition for this behavior is that the model finds a roughly linear/polynomial/exponential space that fits some of the benchmarks, but then fails to capture the complex nature of contention on system resources to accurately model all workloads.

Figure 7 shows the prediction error across models for the CPU power. These results are even more striking. For workloads that lightly load the system (`Sleep`, low-rate `memcached`, etc.) and workloads that stress very specific components (`StressAppTest`, `cpuload`, etc.), the prediction is poor. This is particularly concerning because previous work shows that most production systems are run at low utilization levels [3]. Hence, one might hope that prediction is much better at idle or low load (which is a more realistic scenario in production systems). Unfortunately, the error climbs above 80% for most of the models. Thus, we find that all the models we evaluate are limited in their ability to predict the power consumption of workloads on multicore systems.

The metric of mean absolute error indicates instantaneous prediction error. An astute reader might observe that some prediction applications might be concerned with long term averages and that the instantaneous errors might balance out. While time averaging can reduce the overall impact, many of the benchmarks experience one-sided errors and the models systematically overpredict for some and underpredict for others. Furthermore, even though the percentage error is influenced by the actual power magnitude, we find that most benchmarks are systematically mispredicted by 1–6W on average.

We posit that one of the factors that contributes in a significant way to the poor prediction performance is the increasing presence of hidden power states. In the present case, there are several resources shared across

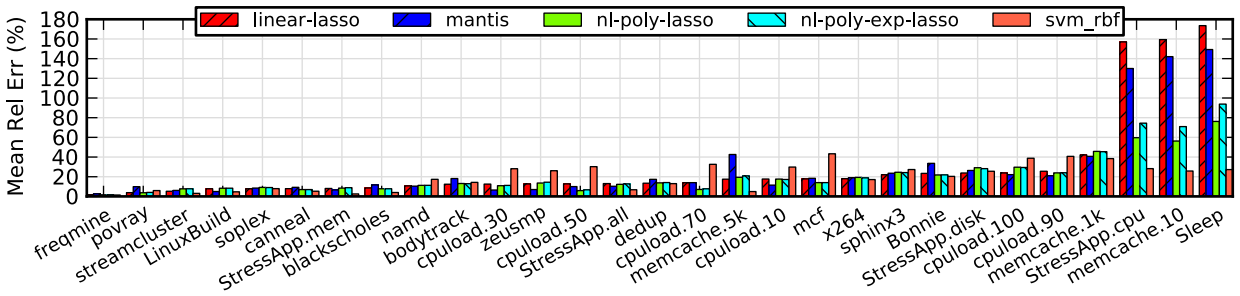


Figure 7: Modeling accuracy for CPU power (Multi-core). Mean relative error is 10–14% across workloads, but as high as 150% for some workloads on the right.

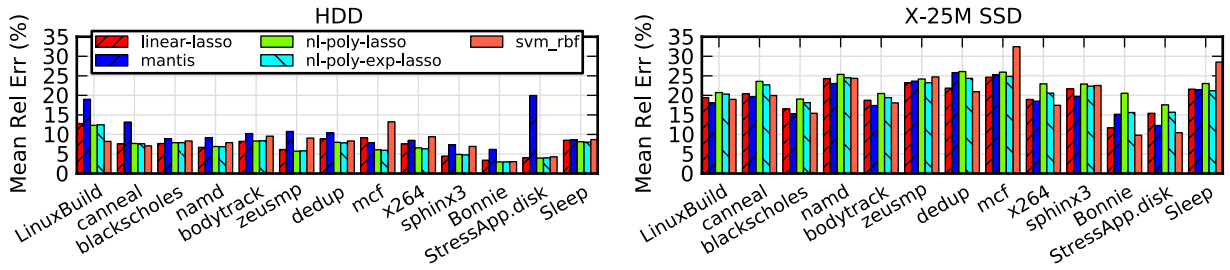


Figure 8: Prediction error with hard disk (left) and newer technology SSD (right).

the cores (L2 caches, for one), which lead to resource contention between the cores causing bottlenecks in processing. However, this very low level behavior is not captured in any of the exported features, and consequently does not make its way into the model. Since we cannot observe the unexposed CPU state to understand what is really happening, in the following section we use the increased internal complexity of SSDs vs. hard drives, instead, to demonstrate degradation in modeling due to hidden power states.

6.3 Hidden states

An important concern when modeling individual hardware components is whether the model, or the inputs to the model, capture all relevant aspects of the component's operation. This derives from the tension between the increasing complexity of the component, and hiding state to present a simple and consistent interface to the OS. If the component incorporates optimizations and circuitry that affect its power draw without varying any of the counters and states that it is exporting externally, then the model, which relies completely on the externally visible features, is likely to fail. As a case in point, there is anecdotal evidence to suggest that newer processors aggressively optimize for power by turning off functional blocks inside the CPU that are not being used, or doing clock gating at a finer granularity than what is exposed on the C-states (neither of which can be easily observed or inferred by software). Another example of this phenomenon can be seen in modern SSD drives: these include a number of performance and robustness optimizations (e.g. wear leveling, page re-writing, etc.). While these complexities are well known [8], they are not exposed via the SATA statistics, as evidenced by low mutual information with the power values. Thus, the power consumed by a given write may have more to do with the hidden state of the device than with the write itself.

To explore this systematically, we ran the same benchmarks on the same platform but with two different hard disks. The first was a conventional 2.5" WD Caviar Black 10K-RPM HDD, and the second was a newer Intel X-25M SSD. Power measurements on the drives show similar power ranges: 0.9–3.6W for the traditional platter based HDD, and 1.2–3.6W for the newer technology SSD. Note that for both disks the features that are recorded are identical, and attempt to capture the amount of work done by the OS in writing/reading from the disk.

Figure 8 shows the prediction errors for both the drives. The high level takeaway is that the error is consistently larger for the SSD than it is for the traditional drive. Specifically, we see that across the set of benchmarks, the model predictions are off by around 7% in the case of the conventional HDD, while they are off by approximately 15% for the SSD drive. Since the features collected and examined in each case are the same, the prediction errors are clearly caused by internal state

changes in the SSD that are reflected in the power draw, but not exposed in the features being tracked. This augurs poorly for power prediction models, given that hardware complexity continues to grow by leaps and bounds.

7 Discussion

While our results indicate that the modeling techniques we study suffer from significant prediction error, it is natural to ask 1) whether this error is in any way fundamental, or could be overcome with more sophisticated techniques, and 2) how useful the resulting predictions might be for particular dynamic power management (DPM) applications. We discuss both issues in this section. First, we present anecdotal evidence that the inherent variability between identical hardware components is likely to introduce a basic error term to any modeling based approach that cannot be solved by adding complexity into the model. Second, we discuss how useful currently achievable levels of accuracy can be to DPM systems.

7.1 Variability

Power modeling is based upon an underlying presumption that the error characteristics of the model do not change over time or over instances of the platform. That is, the model can be generated as a one-time operation by training on a specific platform instance, and the model can be used to predict the power consumption for any *other* instance of platform with the same specifications. While minor variations in manufacturing parts is a given, historically it has not significantly affected the operating characteristics of the platform and processors.

However, the increasing complexity of modern hardware, with staggering amounts of circuitry being stuffed into ever smaller packages exaggerates the variations significantly. These variations lead to variability in power consumption both in the active and standby modes. Furthermore, the variability is only exacerbated by aging and differences in operating environment. Recent work shows that in an 80-core general purpose chip designed by Intel, the frequency, which is directly co-related with the power consumption, of individual cores varied by as much as 25–50% for different operating voltages [13]. Research in embedded systems has shown that multiple instances of a Cortex M3 micro-controller can vary in sleep power consumption by as much as 5× [38]. Recent work has also demonstrated that the performance and power consumption of flash memory chips varies widely based on age, wear and write location [8].

This level of variability raises questions about the ability of power models to generalize over identical systems because they do not actually perform identically. Errors in the power model are amplified by variations across

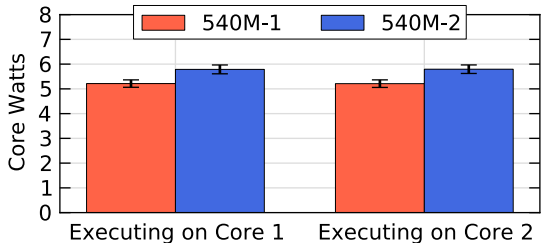


Figure 9: Variability in power consumption measured across two CPUs of the exact same type: Intel Core i5-540M. Core 1 shows a 11% variability between the two processors, while Core 2 shows 11.2% variability. Measurements are averaged across ten runs for each case, with standard deviation marked.

different instances. Using our measurement platform we see significant differences between two identical Core i5-540M processors. Figure 9 illustrates the measured power consumption of the two processors (540M-1 and 540M-2) running our `cpuload` benchmark pinned to either Core 1 or Core 2 using Linux `cpu_sets`. We report CPU power on a single platform so that everything—the mainboard, memory, benchmark, measurement infrastructure, and the operating environment(temperature)—but the processor is constant.

As can be observed from Figure 9, the CPU power consumption when executing on Core 1 for 540M-2 is 11% higher than when using 540M-1, and is similarly 11.2% higher when executing on Core 2. Note that the power consumption is averaged across ten runs on the individual cores (Core 1 or Core 2) for the two processors (540M-1 or 540M-2). We also report the standard deviation in Figure 9 which is measured to be less than 3.1% in all cases. Thus, if one of the models from Section 5 were trained on 540M-1 and applied to workloads executing on 540M-2, then a mean prediction error of 10% could translate into a 23% error and 20% prediction error translates into a 34% prediction error. Given the undeniable trend toward more complex—and therefore inherently variable—components such as processors, this fundamental accuracy gap seems likely to continue to grow. Hence, power instrumentation may be the only choice for accurate power characterization.

7.2 Implications for DPM

As discussed previously, the level of accuracy and the granularity (i.e., which subsystems are characterized) required for dynamic power management is strongly tied to the particular application domain. In some cases, reducing energy might be possible with only a coarse grained and approximate power consumption estimate. In other cases, the application is likely to need a higher degree of accuracy than modeling can currently provide.

A promising way to save energy on a computing platform is by scheduling computation more optimally. This could be done by migrating threads to different cores on the same socket or on different sockets (power gating is often done at the socket level, so using additional cores on the same socket has a very small cost). The processing cores available on a platform may be homogeneous (all derived from identical parts) or heterogeneous (from disparate parts and even architectures).

When there are a multiplicity of processing cores available, we expect the power cost to be quite different, and characterizing the power for each of these cores is critical when deciding to migrate computation. As seen in Section 5, the power models for subsystems like the CPU can have errors of up to 40%. When the errors dwarf the actual power variations across the cores (and this is a likely scenario when the cores are not architecturally different) it is likely that the mispredictions have an adverse effect. However, when the choice is between heterogeneous components such as between a CPU or a GPU, with significantly different power characteristics, i.e., where the variation might be larger than the model errors, it might still be acceptable to rely on modeling.

In another domain, prior work on mobile devices has shown that dynamically switching between multiple heterogeneous radios, such as WiFi and Bluetooth, can in some cases double battery lifetime [27]. Choosing between different radio alternatives like these with vastly different power characteristics seems straightforward even with very poor accuracy. However, recent work has shown that modern WiFi radios, such as those based on the 802.11n MIMO standard, have many more complex states, each with different power consumption tradeoffs [20]. Accurate component-level power characterization will therefore be essential to make optimal decisions on which radio interface or computational unit—and in which mode—to use.

Finally, we note that power-aware resource scheduling is not limited to resources within the same platform. In fact the advent of abundant cloud computing resources has accelerated research into systems, such as MAUI [12], that can use both local computation (on mobile devices) and also execute code remotely in the cloud whenever needed. Currently these systems operate under the assumption that servers in the cloud are always-powered and, hence, their energy costs are not as important as those of battery-powered mobile devices. These systems would benefit significantly from detailed power characterization on the local mobile device as well as the servers in the cloud. Using this information, the policy decisions on when to execute code locally or remotely can be more informed and therefore more optimal. The absolute amounts of energy being considered (i.e., the execution of a single function call) in code offload scenarios, however, are fairly small, so high degrees of accuracy seem essential.

8 Conclusion

The models we consider are able to predict total system power reasonably well for both single core (1–3% mean relative error) and multi-core scenarios (2–6% mean relative error), particularly when the base power of the system is high. However for predicting subsystem power, we show that linear regression based models often perform poorly (10–14% mean relative error, 150% worse case error for the CPU) and more complex non-linear models and SVMs do only marginally better. The poor subsystem power modeling is due to increased system and device complexity and hidden power states that are not exposed to the OS. Furthermore, our measurements show surprisingly high variability in processor power consumption, for the same configuration across multiple identical dies, highlighting the fundamental challenges with subsystem power modeling. Looking forward, while modeling techniques may suffice for some DPM applications, our results motivate the need for pervasive, low-cost ways of measuring instantaneous subsystem power in commodity hardware.

Acknowledgments

We wish to thank Bharathan Balaji for his help with the power measurement setup and Lawrence Saul for his advice regarding the applicability of various machine learning techniques. We also thank our shepherd, Kai Shen, and the anonymous reviewers for their comments and feedback that improved our paper. This work is supported by a grant from Intel and NSF grants CCF-1029783 and CCF/SHF-1018632.

References

[1] Y. Agarwal, S. Savage, and R. Gupta. SleepServer: A Software-Only Approach for Reducing the Energy Consumption of PCs within Enterprise Environments. In *USENIX ATC '10*, 2010.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, 2009.

[3] L. A. Barroso and U. Hölze. The Case for Energy-Proportional Computing. *IEEE Computer*, 2007.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] W. L. Bircher and L. K. John. Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events. *2007 IEEE International Symposium on Performance Analysis of Systems and Software*, 2007.

[6] W. L. Bircher and L. K. John. Predictive Power Management for Multi-Core Processors. In *WEED*, 2010.

[7] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. In *USENIX ATC*, 2010.

[8] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-intensive Applications. *ACM SIGPLAN Notices*, 44(3):217–228, 2009.

[9] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[10] H. Chen, S. Wang, and W. Shi. Where Does the Power Go in a Computer System: Experimental Analysis and Implications, 2010. Technical Report MIST-TR-2010-004.

[11] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley-Interscienc, NY, USA, 1991.

[12] E. Cuervo, A. Balasubramanian, D.-K. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: Making Smartphones Last Longer with Code Offload. In *MobiSys '10*. ACM, 2010.

[13] S. Dighe, S. Vangal, P. Aseron, S. Kumar, T. Jacob, K. Bowman, J. Howard, J. Tschanz, V. Erraguntla, N. Borkar, et al. Within-die Variation-Aware Dynamic-Voltage-Frequency Scaling Core Mapping and Thread Hopping for an 80-core Processor. In *Proceedings of ISSCC*. IEEE, 2010.

[14] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan. Full-System Power Analysis and Modeling for Server Environments. *MOBS*, 2006.

[15] K. Flautner, S. K. Reinhardt, and T. N. Mudge. Automatic Performance Setting for Dynamic Voltage Scaling. In *Proceedings of MobiCom '01*, 2001.

[16] J. Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, 2004.

[17] R. Fonseca, P. Dutta, P. Levis, and I. Stoica. Quanto: Tracking Energy in Networked Embedded Systems. In *OSDI*, 2008.

[18] Glmnet for matlab. `http://www-stat.stanford.edu/~tibs/glmnet-matlab/`.

[19] R. K. Gupta, S. Irani, and S. K. Shukla. Formal methods for Dynamic Power Management. In *Proceedings of ICCAD '03*, 2003.

[20] D. Halperin, B. Greenstein, A. Sheth, and D. Wetherall. Demystifying 802.11n Power Consumption. In *HotPower*, 2010.

[21] A. Kansal, F. Zhao, J. Liu, N. Kothari, and A. Bhattacharya. Virtual Machine Power Metering and Provisioning. In *Proceedings of the 1st ACM Symposium on Cloud computing (SOCC '10)*. ACM, 2010.

[22] R. Koller, A. Verma, and A. Neogi. WattApp: An Application Aware Power Meter for Shared Data Centers. In *ICAC*, 2010.

[23] K. Li, R. Kumf, P. Horton, and T. Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 22–22, 1994.

[24] T. Li and L. K. John. Run-time Modeling and Estimation of Operating System Power Consumption. In *SIGMETRICS*, San Diego, June 2003.

[25] Y. Lu, E. Chung, T. Simunic, G. De Micheli, and L. Benini. Quantitative Comparison of Power Management Algorithms. In *Proceedings of DATE '00*, 2000.

[26] R. Nathuji and K. Schwan. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *SOSP*. ACM, 2007.

[27] T. Pering, Y. Agarwal, R. Gupta, and R. Want. CoolSpots: Reducing the Power Consumption of Wireless Mobile Devices with Multiple Radio Interfaces. In *MobiSys*, 2006.

[28] T. Pering, T. Burd, and R. Brodersen. Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System. In *Power Driven Microarchitecture Workshop, attached to ISCA98*, 1998.

[29] Q. Qiu, Q. Qu, and M. Pedram. Stochastic Modeling of a Power-Managed System-construction and Optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2002.

[30] V. J. Reddi, B. C. Lee, T. Chilimbi, and K. Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. In *ISCA*, 2010.

[31] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A Comparison of High-Level Full-System Power Models. In *HotPower*, San Diego, December 2008.

[32] N. J. Sanders. Stressapptest. `http://code.google.com/p/stressapptest/`.

[33] K. Shen, M. Zhong, S. Dwarkadas, and C. Li. Hardware counter driven on-the-fly request signatures. *ASPLOS*, 2008.

[34] SpecCPU2006. `http://www.spec.org/cpu2006`.

[35] T. Stathopoulos, D. McIntire, and W. J. Kaiser. The Energy Endoscope: Real-time Detailed Energy Accounting for Wireless Sensor Nodes. In *IPSN*, 2007.

[36] N. Tolia, Z. Wang, M. Marwah, C. Bash, and P. Ranganathan. Delivering Energy Proportionality with Non Energy-Proportional Systems–Optimizing the Ensemble. In *HotPower*, San Diego, December 2008.

[37] X. Wang and Y. Wang. Coordinating Power Control and Performance Management for Virtualized Server Clusters. *TPDS*, 2010.

[38] L. F. Wanner, R. Balani, S. Zahedi, C. Apte, P. Gupta, and M. B. Srivastava. Variability-Aware Duty Cycle Scheduling in Long Running Embedded Sensing Systems . In *Proceedings of Design, Automation and Test in Europe 2011 (DATE)*, 2011.

[39] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *ASPLOS*, 2002.

Victim Disk First: An Asymmetric Cache to Boost the Performance of Disk Arrays under Faulty Conditions

Shenggang Wan, Qiang Cao, Jianzhong Huang,*
Siyi Li, Xin Li, Shenghui Zhan, Li Yu, Changsheng Xie
Huazhong University of Science & Technology
Wuhan, China 430074
**Corresponding Author: Caoqiang@hust.edu.cn*

Xubin He
Electrical & Computer Engineering
Virginia Commonwealth University
Richmond, VA 23284, USA
xhe2@vcu.edu

Abstract

The buffer cache plays an essential role in smoothing the gap between the upper-level computational components and the lower-level storage devices. A good buffer cache management scheme should be beneficial to not only the computational components, but also to the storage components by reducing disk I/Os. Existing cache replacement algorithms are well optimized for disks in normal mode, but inefficient under faulty scenarios, such as a parity-based disk array with faulty disk(s).

To address this issue, we propose a novel asymmetric buffer cache replacement strategy, named Victim (or faulty) Disk(s) First (VDF) cache, to improve the reliability and performance of a storage system consisting of a buffer cache and disk arrays. The basic idea is to give higher priority to cache the blocks on the faulty disks when the disk array fails, thus reducing the I/Os directed to the faulty disks.

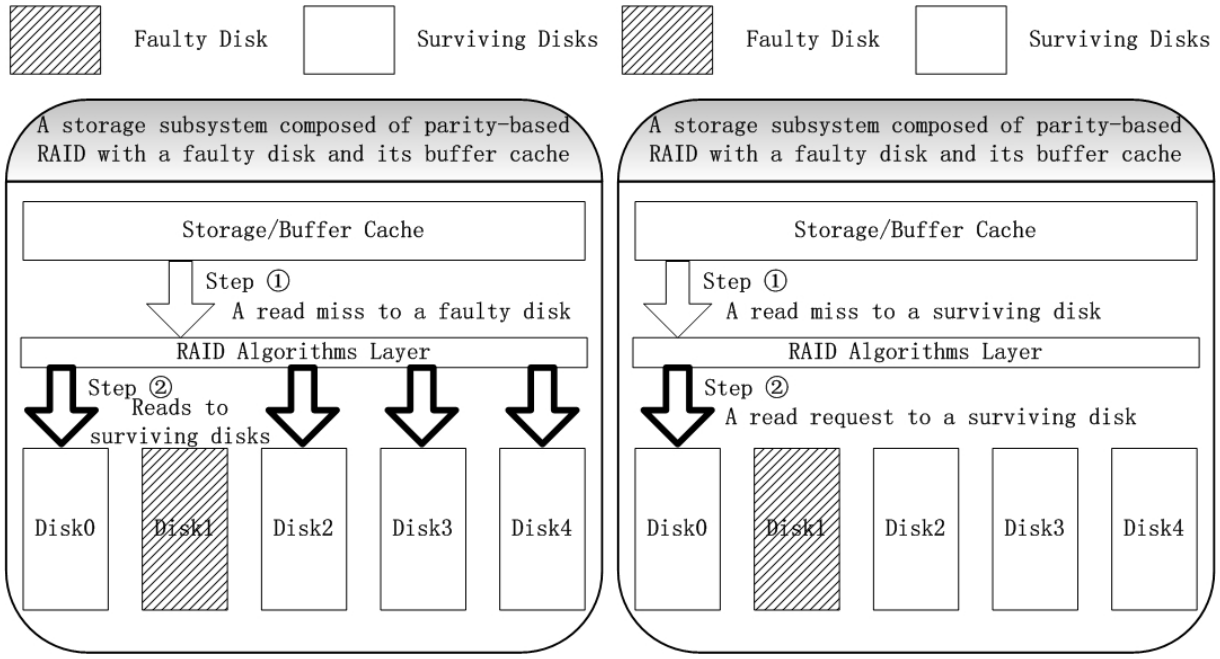
To verify the effectiveness of the VDF cache, we have integrated VDF into two popular cache algorithms LFU and LRU, named VDF-LFU and VDF-LRU, respectively. We have conducted extensive simulations as well as a prototype implementation. The simulation results show that VDF-LFU can reduce disk I/Os to surviving disks by up to 42.3% and VDF-LRU can reduce those by up to 36.2%. Our measurement results also show that VDF-LFU can speed up the online recovery by up to 46.3% under a spare-rebuilding mode with online reconstruction, or improve the maximum system service rate by up to 47.7% under a degraded mode without a reconstruction workload. Similarly, VDF-LRU can speed up the online recovery by up to 34.6%, or improve the system service rate by up to 28.4%.

1 Introduction

To reduce the number of I/O requests to the low level storage device, such as disk arrays, a cache is widely used and many cache algorithms exist to hide the long disk latencies. These cache algorithms work well for disk arrays under normal fault-free mode. However, when some disks in a disk array fail, the RAID may still work under this faulty scenario, either in a *spare-rebuilding mode* with online reconstruction or in a *degraded mode* without online reconstruction. The cost of a miss to faulty disks might be dramatically different compared to the cost of a miss to surviving disks. Existing cache algorithms cannot capture this difference because they treat the underlying (faulty or surviving) disks the same.

We take an example as shown in Figure 1, which illustrates two different cache miss situations in a storage subsystem composed of a parity-based RAID with one faulty disk in degraded mode. As shown in Figure 1(a), the missed data resides in the faulty disk. The RAID controller accesses the surviving disks to fetch all data and parity in the same stripe to regenerate the lost data. Therefore, to service one cache miss, several read requests are needed depending on the RAID organization. However, if the missed data is in a surviving disk as shown in Figure 1(b), only one read request to the corresponding surviving disk is generated. Similar situations are observed in spare-rebuilding mode. A simple analysis shows that in a RAID5 system consisting of n disks, when a disk fails, the cost to fetch data from a faulty disk might be $n-1$ times higher than the cost to access data from a surviving disk. This extra disk I/O activity will in turn reduce the effective array bandwidth available for reconstruction or user access.

When a disk array starts online reconstruction, it uses up regular bandwidth. Compared to offline reconstruction, during the process of online reconstruction, the user workflow interferes with the reconstruction workflow. As a result, the online reconstruction duration grows significantly compared to offline reconstruction. Wu et al. [1] point out that, in a heavy user workflow, the duration of online reconstruction would grow as much as 70 times as that of the offline reconstruction. In this



(a) A read miss to a faulty disk might result in several additional read requests to the surviving disks.

(b) A read miss to a surviving disk would result in only one request to the corresponding surviving disk.

Figure 1: Two typical cache-miss situations in a storage subsystem composed of a parity-based RAID in a degraded mode.

case, more requests to the surviving disks, caused by user requests, reduce the available reconstruction bandwidth and lengthen the reconstruction duration, which reduces the reliability of the storage system.

On the other hand, in a degraded mode without a reconstruction workflow, a miss to faulty disks would cause all the surviving data in the same parity chain (stripe in RAID-5) to be read and add additional workflow to surviving disks. With a decreasing serviceability and an increasing user workflow caused by misses to faulty disks, the storage subsystem might be overloaded under a heavy user workflow.

Therefore, in parity-based disk arrays under faulty conditions, a miss to faulty disks is much more expensive than a miss to surviving disks. Based on this observation, we propose an asymmetric buffer cache replacement strategy, named Victim (or faulty) Disk(s) First cache, or VDF for short, to improve the performance of storage subsystem composed of a parity-based disk array and its buffer cache. The basic idea is to design a cache scheme to treat the faulty disks more favorably, or give higher priority to cache the data associated with the faulty disks. The goal of this scheme is to reduce the cache miss directed to the faulty disk, and thus to reduce the I/O requests to the surviving disks overall. Reduced disk I/O caused by the user workflow will (1) improve the performance of the disk array, and (2) allow more bandwidth for online reconstruction which in turn speeds up the recovery, and thus improves the reliability. We make the following four contributions in this paper:

1. We proposed a new metric, *Requests Generation Ratio* or RGR, to capture the disk I/O activities of user workflows on the surviving disks when a storage system is under faulty conditions. This would directly influence the maximum bandwidth for reconstruction in a spare-rebuilding mode and the bandwidth available to user workflows in a degraded mode.

2. We developed a novel cache-replacement scheme, VDF, by giving higher priority to cache the data associated with the faulty disks, to minimize the RGR. VDF is flexible and could be integrated into existing cache algorithms such as LRU and LFU.

3. We conducted extensive simulations to verify the effectiveness of VDF under different workloads. The simulation results show that VDF-LRU can reduce overall disk I/Os to surviving disks by up to 36.2% and VDF-LFU can reduce those by up to 42.3%.

4. We implemented VDF in the Linux software RAID system. As a result, VDF-LFU can speed up the online recovery by up to 46.3% under spare-rebuilding

mode, or improve the maximum system service rate by up to 47.7% under degraded mode. Similarly, VDF-LRU can speed up the online recovery by up to 34.6%, or improve the system service rate by up to 28.4%.

The rest of the paper is organized as follows: Section 2 gives a brief overview of the background information and related work. In Section 3, we describe our new metric, RGR, and the design of VDF. A case study of VDF cache is given in Section 4; we describe integrating VDF into two typical cache-replacement algorithms, LRU and LFU, based on RAID-5. We provide our simulation results of VDF in Section 5, and prototyping and measurement results in Section 6. We conclude our paper and describe future work in Section 7.

2 Background and Related Work

In this section we briefly overview some background materials and related work.

2.1 Optimizations of Disk Arrays under Faulty Conditions

Redundant **A**rrays of **I**ndependent **D**isks **RAID** [2] are popular solutions to provide high performance and reliability for today's storage systems. Depending on its organizations, RAID could prevent data loss incurred by disk failures and even offer online services under faulty conditions. With a faulty disk, these RAIDs would work in a spare-rebuilding mode to support online reconstruction, or in a degraded mode without reconstruction.

RAID can offer continuous online services even in faulty mode. However, the recovery workload and user request can interfere with each other, and lead to longer recovery times. Many solutions are proposed to solve this problem, such as optimizations of data/parity/spare layout [3–7], reconstruction workload [8–12], and user workload [1, 13, 14].

Menon et al. present a method to distribute spares to all disks, which would not only reduce the lost data per disk but also parallelize the reconstruction [4]. Holland et al. [3] propose a trade-off between RAID-1 (mirror) and RAID-5, named parity declustering, to balance the storage efficiency and the recovery performance. Xin et al. use a RUSH-like hash algorithm to evenly distribute data, parity, and spares among the nodes in a distributed environment [5].

The track-based recovery (TBR) [8] algorithm provides a trade-off between block-based recovery and cylinder-based recovery, and balances the user response time and the recovery duration. However, TBR requires much more buffer space compared to block-based recovery. The pipelined recovery (PR) scheme [9] addresses this problem, and significantly reduces the buffer requirements close to that of the block-based recovery algorithms. The disk-oriented recovery (DOR) algorithm [10] rebuilds the array at the disk-level instead of the stripe-level. With this approach, DOR could absorb the bandwidth of the array as much as possible. The popularity-based recovery (PRO) algorithm [11, 12], builds upon the DOR algorithm, further improving the recovery performance by utilizing the spatial locality of user requests.

Two techniques named *redirection of reads* and *piggybacking of writes* [13] are proposed to reduce the user workflow by employing the reconstructed spare disk to absorb parts of the requests to the faulty disk. However, they need to maintain a bitmap in the dedicated cache in the RAID device to record the reconstruction status; as the increasing of disk size, a fine-granularity bitmap would consume too much memory, and increase synchronization costs. For example, a bitmap with granularity of 4KB for a 2TB disk would require 64MB of memory, which limits the use of piggybacking of writes. During the reconstruction, at most a coarse-granularity bitmap could be used only to redirect reads. MICRO [14] achieves improved recovery performance by writing back the in-memory surviving data of the faulty disks into a spare disk first and using a file popularity table to find the hotspot. MICRO treats all the blocks in the cache equally, which is similar to the general cache-replacement algorithms and has the same limitations. WorkOut [1], an array-cache-array method, offloads the write requests and popular read requests to another disk array. As a result, WorkOut speeds up the recovery process and improves the user response time. However, WorkOut requires another disk array to help with the reconstruction and need maintain an addressing translation map, which might be much larger than a fine-granularity bitmap, in the dedicated cache on the RAID device. This suffers from the same problem as redirection of reads and piggybacking of writes.

2.2 Buffer Cache Replacement Algorithms

RAID-based storage systems usually work together with the buffer cache. To improve the efficiency of the buffer cache, researchers have proposed many cache-replacement algorithms, such as LRU [15], LFU, FBR [16], LRU-k [17, 18], 2Q [19], LRFU [20, 21], MQ [22, 23], LIRS [24, 25], ARC [26], DULO [27], DISKSEEN [28] and more. Each cache-replacement algorithm weigh the cached blocks with a different method, such as access interval, access frequency and so on, then decide which cached blocks to evict.

Table 1: Variables and Definitions

Symbols	Definition
C	Total number of blocks in the buffer cache
T	Total number of data blocks in a disk array
B_i	Data block i
p_i	Access probability of each block B_i
MP_i	Miss penalty of each block B_i
BW	Total serviceability of all surviving disks in terms of I/O bandwidth
BW_U	I/O bandwidth available to user workload, or service rate of the system from the user's point of view
BW_R	I/O bandwidth for an online reconstruction workload
RGR	The ratio of the # of requested blocks to surviving disks and the # of requested blocks to buffer cache
Q	Total amount of data from surviving disks to reconstruct faulty disks

The LRU (Least-Recently-Used) algorithm is one of the most popular and effective policies for buffer cache management. When a block needs to be inserted into the cache, the candidate to be evicted is the block which is least recently used. That is to say, the weight of the cached blocks in LRU is its last access timestamp. The block with the smallest last access timestamp is evicted. The LFU (Least-Frequently-Used) algorithm replaces the least frequently used block. In other words, the weight of the cached blocks in LFU is its number of accesses. The block with the smallest number of accesses is evicted. Other algorithms, such as LRU-k, 2Q, LRFU, MQ, LIRS, and ARC, integrate LRU and LFU algorithm together and demonstrate good performance under various scenarios. DULO and DISKSEEN consider both temporal and spatial locality when a block needs to be replaced.

However, the above cache-replacement algorithms work well when the RAID system is under normal operating mode. When some disks in the RAID system fail, it runs under faulty condition, but the buffer cache layer is not aware of the underlying failures in RAID and thus the existing cache algorithms do not work well as explained in Section 1. This motivates us to propose VDF: a cache scheme to treat the faulty disks more favorably, or give a higher priority to cache the data associated with faulty disks. The goal is to reduce the cache misses directed to the faulty disk and thus to reduce the I/O requests to the surviving disks overall. As our VDF only increases the weight of blocks in the faulty disks, theoretically it could work with the above-mentioned general cache-replacement algorithms.

3 Design of VDF

In this section, we propose a new metric to describe the cache efficiency of disk I/O activities. We show how to use it to evaluate disk arrays under faulty conditions, and then we describe our VDF scheme. Before our discussion, we summarize the symbols in Table 1.

3.1 RGR: A New Metric to Evaluate Cache Performance with Various Miss Penalty

Traditional cache-replacement algorithms are essentially evaluations on access probability of cached blocks, based on the assumption that the penalty of each miss at the same level is the same. However, in parity-based RAID with faulty disk(s), the penalty of a miss to the lost data in the faulty disks might be much more expensive than that of a miss to surviving data. Therefore, from the aspect of a RAID device, the buffer cache performance should not be simply evaluated by the traditional metrics such as Hit Ratio or Miss Ratio, particularly when the RAID is under faulty conditions. To address this issue, we propose a new metric called *Requests Generation Ratio* or RGR. This is the ratio of the number of requested blocks to the surviving disks and the number of the requested blocks to buffer cache, to evaluate the cache performance from the view point of a faulty RAID device. RGR represents the disk activities to service an I/O request to the buffer cache. Ideally, if all I/O requests are serviced by the buffer cache, RGR will be 0 (no disk I/Os are generated). For missed I/O requests, RGR will be different depending on the penalty to each underlying disk. For example, in Figure 1(a) the RGR of a miss to the faulty disk is 4 because 4 disk I/Os are generated to service the missed I/O request, and in Figure 1(b), the RGR of a miss to surviving disks is 1.

To calculate the RGR, we assume a parity-based RAID of T data blocks with a buffer cache of C blocks. The access probability of a block B_i is p_i, where $0 \leq i \leq T-1$, with a miss penalty of MP_i in terms of the total requested blocks to surviving disks caused by a miss. From the viewpoint of a certain workload, p_i actually repre-

sents the ratio of the number of request on B_i and the number of total block requests. If block B_i is not referenced in this workload, p_i should be 0. As we have mentioned above, different cache algorithms evaluate p_i with different approaches in runtime environments. If a block is serviced by the cache, the corresponding miss penalty $MP_i = 0$. Therefore, the RGR of the next block request can be described by the following Equation 1.

$$RGR = \sum_{i=0}^{T-1} (p_i \times MP_i) \quad (1)$$

3.2 Using RGR to Evaluate the Cache Efficiency in Faulty Mode

Consider a system composed of a buffer cache and a RAID in faulty mode which service a certain user workload. We have the following symbols. First, the total serviceability of all surviving disks is BW in terms of I/O bandwidth. Second, the unfiltered user workload would take BW_U bandwidth, which is the service rate of the system from a user's perspective. The average RGR of the buffer cache is RGR. Therefore, the filtered user workload should take about $BW_U \times RGR$ bandwidth. Third, all the remaining bandwidth BW_R of all surviving disks could be utilized for reconstruction. Lastly, the total amount of surviving data for reconstruction is Q. Equation 2 describes the relationships among BW, BW_U, RGR, and BW_R.

$$BW = BW_U \times RGR + BW_R \quad (2)$$

We first consider the spare-rebuilding mode. The surviving disks would suffer from more requests as explained in Section 1. It means that the I/O bandwidth available for reconstruction on the surviving disks would be less than the I/O bandwidth for reconstruction on the spare disk. The total amount of requested data for reconstruction on each disk (including the surviving disks and the spare disks) is the same. Therefore, to the online recovery process, the I/O bandwidth for reconstruction on the surviving disks is the bottleneck. The reconstruction duration RD could be described with Equation 3.

$$RD = \frac{Q}{BW - BW_U \times RGR} \quad (3)$$

From Equation 3, we can find that, if Q, BW, and BW_U are fixed, with the decreasing RGR, the reconstruction duration RD (and thus MTTR) decreases. Therefore, to minimize the MTTR, we should minimize the RGR.

We next consider the degraded mode without reconstruction. Each surviving disk would suffer from the extra requests caused by the access to faulty disks. The filtered user workload should not exceed the total serviceability of all surviving disks. In another words, the maximum unfiltered user workload BW_U should not exceed $\frac{BW}{RGR}$. Therefore, we should minimize the RGR to maximize the system serviceability, which is described with a maximum BW_U.

From the above discussion, we notice that compared to the traditional metrics on cache evaluation, such as miss ratio, RGR is a useful metric to demonstrate two important indicators of a faulty disk array more clearly and directly. One is the reconstruction time which is directly related to MTTR and affects the system reliability. The other is the throughput that indicates the performance of the storage system.

3.3 VDF Cache

Based on the above analysis, we propose our VDF cache aiming at reducing the RGR for parity-based RAID under faulty conditions, either to enhance the system reliability by speeding up the reconstruction process in spare-rebuilding mode, or to improve the system performance by increasing the system serviceability in degraded mode without reconstruction workloads. As it operates at the buffer cache level, VDF is practical and does not suffer from the same problems of the small dedicated cache in a RAID controller.

Cache-replacement algorithms are essentially evaluations on access probability of cached blocks. Once a miss occurs, typically a block should be evicted from cache, and the missed block would be loaded to the free space. General replacement algorithms evict the block with the smallest access probability to reduce the total access probability of the remaining blocks out of buffer cache. However, to minimize the RGR, the eviction of a block should not only be determined by the access probability but also by the miss penalty. In our VDF cache, we adopt the same evaluation approach of access probability p_i for each cached block as the general cache. Furthermore, the miss penalty MP_i of each block is evaluated with the requested blocks to the surviving disks of this block; the block with the minimum product of $p_i \times MP_i$ is evicted from the cache to minimize the RGR.

4 A Case Study of VDF

To verify the effectiveness of VDF, we apply VDF in a RAID-5 system of n disks where one disk fails. We focus on read operations for two reasons: first, in many applications, users are typically sensitive to read latency, particularly in a disk array under faulty conditions; second, in many storage systems, independent non-volatile memory is deployed as a write cache to enhance the reliability and this cache uses a dedicated write cache algorithm.

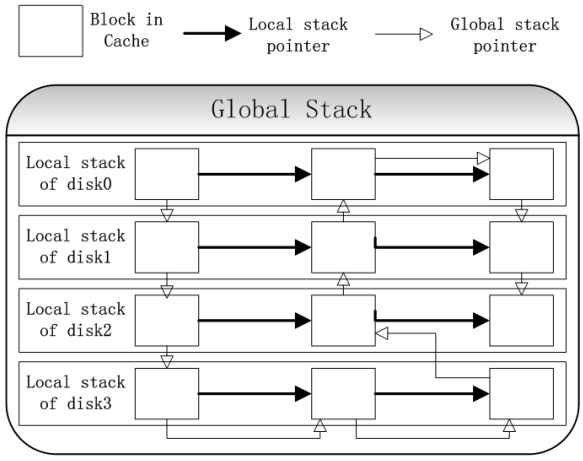


Figure 2: VDF implementation with two types of stacks.

Algorithm 1: VDF-LRU for RAID-5 with n disks

```
Input: The request stream x_1, x_2, x_3, ..., x_i, ...
VDF_LRU_Replace(x_i){
/*For every i ≥ 1 and any x_i, one and only one of the following cases
must occur.*/
if x_i is in LS_k, 0 ≤ k < n then
    /*A cache hit has occurred.*/
    Update TS of x_i, by TS = GTS;
    Move x_i to the heads of LS_k and GS.
else
    /*A cache miss has occurred.*/
    if Cache is full then
        foreach block at the bottom of LS_j, 0 ≤ j < n do
            if LS_j is a corresponding stack to a faulty disk then
                Its weight W=GTS − TS;
            else
                Its weight W=(GTS − TS) ∗ (n − 1);
        Delete the block with maximum W to obtain a free block;
    else
        /*Cache is not full.*/
        Get a free block.
    Load x_i to the free block.
    Update TS of x_i, by TS = GTS;
    Add x_i to the heads of GS and the corresponding LS.
Update GTS, by GTS = GTS + 1;
}
```

4.1 Integrating VDF into LRU and LFU

Although VDF cache can cooperate with caches at other levels by adjusting the miss penalty of blocks, for demonstration purposes we just consider a one-level buffer cache above the disk array. Therefore, the miss penalty of blocks on the faulty disk would be $n-1$ in our following discussion, which means one cache miss to the faulty disk will result in $n-1$ I/O requests to the surviving disks. To integrate VDF with any cache algorithm, the access probability should be evaluated with a quantitative approach. Different cache-replacement algorithms evaluate the access probability of blocks using different approaches. Most existing cache management algorithms can be categorized into LRU-like and LFU-like algorithms. In LRU-like algorithms, the weight of blocks is often evaluated by the access time interval. As it is costly to record the real access timestamp, a simple alternative is to record the access sequence number, and use the reciprocal of the interval access sequence number as the access probability. This approach is widely used in many LRU-like algorithms. In LFU-like algorithms, the weight of blocks is majorly evaluated by the access frequency. Thus, to integrate VDF into these LFU-like algorithms, it needs only to keep the original evaluation approach. Furthermore, different from the access sequence number, the access frequency of two blocks might be the same. Therefore, in VDF based LFU-like algorithms, the access sequence number is also employed for choosing which block to evict with the same access frequency. In VDF cache the access probability of a block would not be the absolute but the relative value, because both the reciprocal of interval of access sequence number and the access frequency are actually the relative values.

The conversion from the original cache algorithms to the VDF-based algorithms should be smooth, because VDF takes effect in faulty mode. In other words, the buffer cache should be managed with the original algorithms in fault-free mode, and the VDF policy becomes effective when disk failures occur. Thus, a smooth runtime conversion between the original algorithm and the VDF-based algorithm is needed, which is quite different from the general cache algorithms. Therefore, in VDF-based algorithm, we employ two types of stacks to achieve the smooth runtime conversion: one is the global stack (GS) which is similar to the stack in a general algorithm such as global LRU stack, and the other is the local stack (LS) holding the blocks on the same disk in cache. All blocks should be in two types of stacks concurrently as shown in Figure 2. When the system works in fault-free mode, it evicts the block with the smallest weight at the bottom of the GS stack. Once a disk array drops to a faulty mode, it evicts the block with the smallest weight at the bottom of each LS stack instead of evicting the block at the bottom of the GS stack.

4.2 Detailed Description of VDF-LRU and VDF-LFU

Detailed descriptions of VDF-LRU and VDF-LFU for n-disk RAID-5 are given in Algorithm 1 and Algorithm 2, respectively, using the variables summarized in Table 2.

5 Simulation Results and Analysis

To evaluate the effectiveness of VDF, we conducted simulations under three typical workloads: SPC-1-web, LM-TBE, and DTRS.

Table 2: Variants in VDF and Explanation

Variants	Explanation
x	A block request to buffer cache
LS	The local stack holding the blocks on one certain disk in the buffer cache
GS	The global stack holding all the blocks on all the disks in the buffer cache
n	The total number of disks including the faulty disk and surviving disks
TS	The timestamp of a block: records the access sequence number
F	The access frequency of a block
GTS	The global timestamp: it is equal to the timestamp of currently accessed block
W	The weight of a block

Algorithm 2: VDF-LFU for RAID5 of n disks

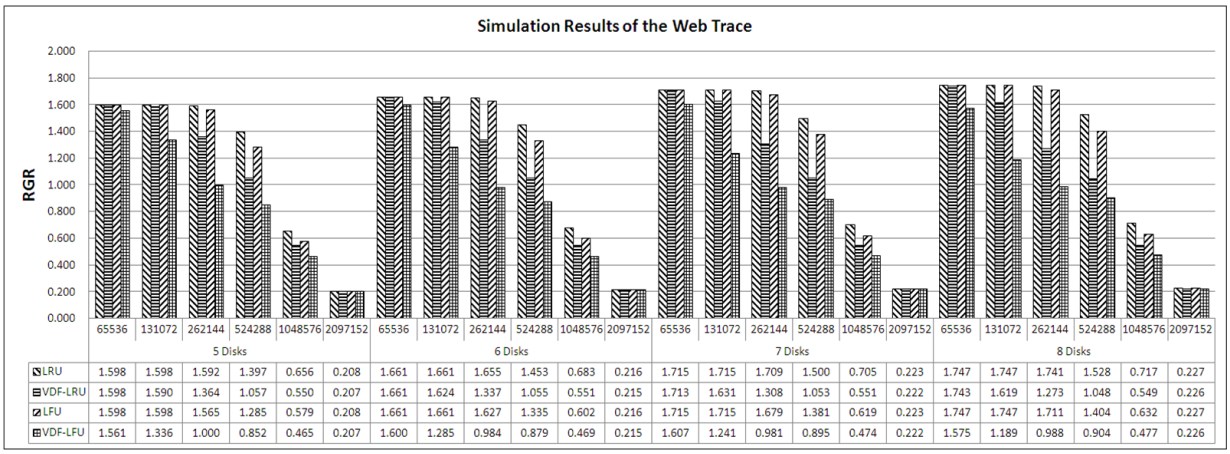
```
Input: The request stream x_1, x_2, x_3, ..., x_i, ...
VDF_LFU_Replace(x_i){
/*For every i ≥ 1 and any x_i, one and only one of the following cases
must occur.*/
if x_i is in LS_k,0 ≤ k < n then
    /*A cache hit has occurred.*/
    Update F and TS of x_i, by F = F + 1;
    Move x_i to right place of LS_k and GS according to F and TS.
else
    /*A cache miss has occurred.*/
    if Cache is full then
        foreach block at the bottom of LS_j, 0 ≤ j < n do
            if LS_j is a corresponding stack to a faulty disk then
                Its weight W=F * (n − 1);
            else
                Its weight W=F;
        Delete the block with minimum W and GTS − TS to obtain
        a free block;
    else
        /*Cache is not full.*/
        Get a free block.
    Load x_i to the free block.
    Initialize the frequency F and TS of x_i, by F = 1 and
    TS = GTS;
    Move x_i to right place of LS_k and GS according to F and TS.
Update GTS, by GTS = GTS + 1;
}
```

SPC-1-web, a trace used in the SPC-1 benchmark suites, was collected in a search engine, which is widely used in the evaluation of storage systems [1,11,14]. LM-TBE and DTRS are provided by Microsoft Corporation collected in 2008. The LM-TBE trace was collected in back-end servers supporting a front-end Live Maps application. The DTRS trace was collected in a file server accessed by more than 3000 users to download various daily builds of Microsoft Visual Studio. Both traces were taken in a period of 24 hours and broken into pieces with 1-hour intervals [29]. We choose only the piece with most intensive I/O activities. For fairness and simplicity, we consider only the read operations and all block sizes are 4KB. We report RGR of LRU, LFU, VDF-LFU, and VDF-LRU under these workloads as shown in Figures 3, 4, and 5, respectively.

Our simulator, named VDF-Sim, is written in C and the source code is approximately 3000 lines. It slices/splits the trace records into block requests as the input. Data blocks in a stack or blocks with the same hash values are linked via double circular lists. For a certain block in our simulator, we record its logical offset as the unique ID since the disk array is transparent to the upper level systems such as a file system. According to the data/parity distribution of the disk array and the logical offset of a block, it is easy to identify on which disk the block resides. The arriving timestamps of the requests are also recorded to evaluate p_i as we mentioned in Section 4 and to generate the misses trace used in the prototype discussed in the next section.

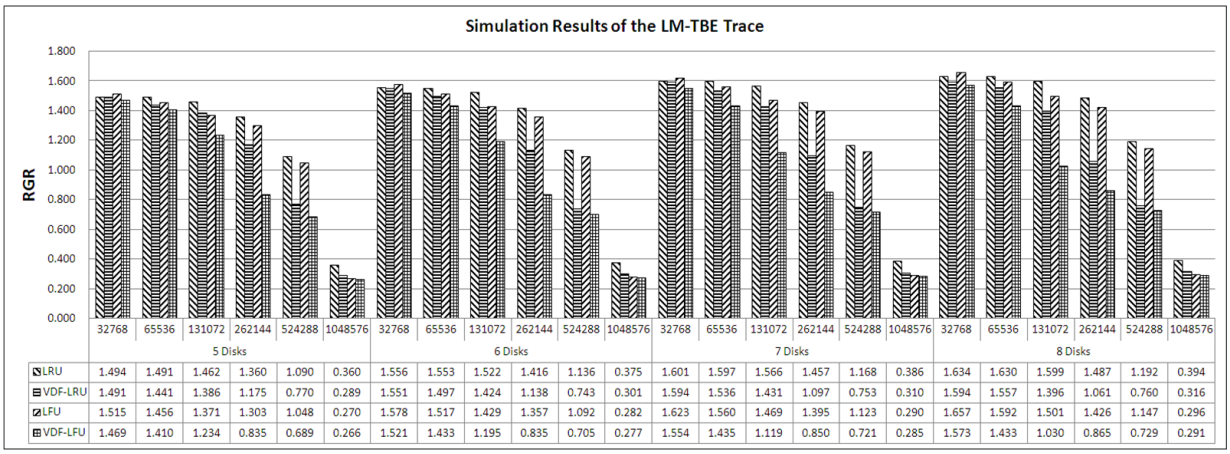
The results show that, compared to the original LRU and LFU algorithms, VDF optimized algorithms achieved better performance consistently by reducing the RGR. Compared to LRU, VDF-LRU reduces the RGR by up to 31.4%, 36.2%, and 22.7% under SPC-1-web, LM-TBE, and DTRS traces, respectively. Compared to LFU, VDF-LFU reduces the RGR by up to 42.3%, 39.4%, and 24.4%, respectively.

We find that the efficiency of VDF grows with the increased number of disks under the same number of cache-resident blocks in most cases. The efficiency of VDF is more significant with a moderate number of cache-resident blocks than that with a too small or too large number of cache-resident blocks. This can be explained as follows. The cache-resident blocks of a faulty disk in the original algorithm would occupy $1/n$ cache space with total n disks. With the fixed cache-resident blocks and the increased n, the number of cache-resident blocks of the faulty disk would be smaller. From the aspect of cache management, the impact of the marginal utility of blocks on hit ratio tends to decrease with the increased cache size. For example, adding $P1$ blocks to a cache with $P2$ blocks might improve the hit ratio with a larger gain compared to adding $P1$ blocks to cache with $P3$ blocks when $P2 < P3$. Thus, the marginal utility of blocks would be more obvious with more disks and thus the efficiency of VDF grows accordingly. However, if the number of cache-resident blocks is too small, it is hard to find hot blocks even with an extended period due



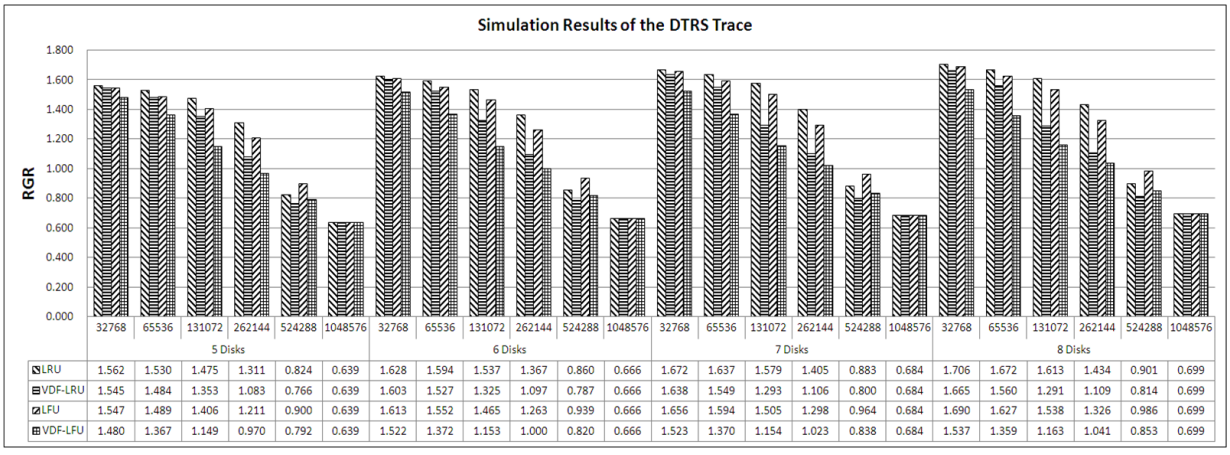
	65536	131072	262144	524288	1048576	2097152	65536	131072	262144	524288	1048576	2097152	65536	131072	262144	524288	1048576	2097152	65536	131072	262144	524288	1048576	2097152
	5 Disks						6 Disks						7 Disks						8 Disks					
LRU	1.598	1.598	1.592	1.397	0.656	0.208	1.661	1.661	1.655	1.453	0.683	0.216	1.715	1.715	1.709	1.500	0.705	0.223	1.747	1.747	1.741	1.528	0.717	0.227
VDF-LRU	1.598	1.590	1.364	1.057	0.550	0.207	1.661	1.624	1.337	1.055	0.551	0.215	1.713	1.631	1.308	1.053	0.551	0.222	1.743	1.619	1.273	1.048	0.549	0.226
LFU	1.598	1.598	1.565	1.285	0.579	0.208	1.661	1.661	1.627	1.335	0.602	0.216	1.715	1.715	1.679	1.381	0.619	0.223	1.747	1.747	1.711	1.404	0.632	0.227
VDF-LFU	1.561	1.336	1.000	0.852	0.465	0.207	1.600	1.285	0.984	0.879	0.469	0.215	1.607	1.241	0.981	0.895	0.474	0.222	1.575	1.189	0.988	0.904	0.477	0.226

Figure 3: Simulation results under the SPC-1-web trace. The number of disks ranges from 5 to 8, and number of cache blocks varies from 64K to 2M with the block size of 4KB.



	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576
	5 Disks						6 Disks						7 Disks						8 Disks					
LRU	1.494	1.491	1.462	1.360	1.090	0.360	1.556	1.553	1.522	1.416	1.136	0.375	1.601	1.597	1.566	1.457	1.168	0.386	1.634	1.630	1.599	1.487	1.192	0.394
VDF-LRU	1.491	1.441	1.386	1.175	0.770	0.289	1.551	1.497	1.424	1.138	0.743	0.301	1.594	1.536	1.431	1.097	0.753	0.310	1.594	1.557	1.396	1.061	0.760	0.316
LFU	1.515	1.456	1.371	1.303	1.048	0.270	1.578	1.517	1.429	1.357	1.092	0.282	1.623	1.560	1.469	1.395	1.123	0.290	1.657	1.592	1.501	1.426	1.147	0.296
VDF-LFU	1.469	1.410	1.234	0.835	0.689	0.266	1.521	1.433	1.195	0.835	0.705	0.277	1.554	1.435	1.119	0.850	0.721	0.285	1.573	1.433	1.030	0.865	0.729	0.291

Figure 4: Simulation results under the LM-TBE trace with various numbers of disks and cache blocks. The block size is 4KB.



	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576	32768	65536	131072	262144	524288	1048576
	5 Disks						6 Disks						7 Disks						8 Disks					
LRU	1.562	1.530	1.475	1.311	0.824	0.639	1.628	1.594	1.537	1.367	0.860	0.666	1.672	1.637	1.579	1.405	0.883	0.684	1.706	1.672	1.613	1.434	0.901	0.699
VDF-LRU	1.545	1.484	1.353	1.083	0.766	0.639	1.603	1.527	1.325	1.097	0.787	0.666	1.638	1.549	1.293	1.106	0.800	0.684	1.665	1.560	1.291	1.109	0.814	0.699
LFU	1.547	1.489	1.406	1.211	0.900	0.639	1.613	1.552	1.465	1.263	0.939	0.666	1.656	1.594	1.505	1.298	0.964	0.684	1.690	1.627	1.538	1.326	0.986	0.699
VDF-LFU	1.480	1.367	1.149	0.970	0.792	0.639	1.522	1.372	1.153	1.000	0.820	0.666	1.523	1.370	1.154	1.023	0.838	0.684	1.537	1.359	1.163	1.041	0.853	0.699

Figure 5: Simulation results under the DTRS trace with various numbers of disks and cache blocks. The block size is 4KB.

to the large access interval. On the other hand, if the number of cache-resident blocks is too large, most of the requested blocks from the faulty disks would be cached, and the marginal utility of blocks becomes insufficient.

We also find that the VDF strategy becomes more efficient with LFU than LRU under the three workloads. One possible reason is that the temporal locality of these traces is weak as they are server-end traces and already filtered by upper level caches. Thus, the *Stack Depth Distribution* property of these traces is weak. As a result, the *Independent Reference Model* property of these traces would be relatively improved. Although we improve the weight of the blocks on faulty disk to $n-1$ times in both VDF-LRU and VDF-LFU, the efficiency is not the same. Mostly, the caching duration of blocks from a victim disk in VDF-LFU is longer than that in VDF-LRU, especially when the total number of cache-resident blocks is not large. Therefore, VDF-LFU works better than VDF-LRU in these traces in most cases.

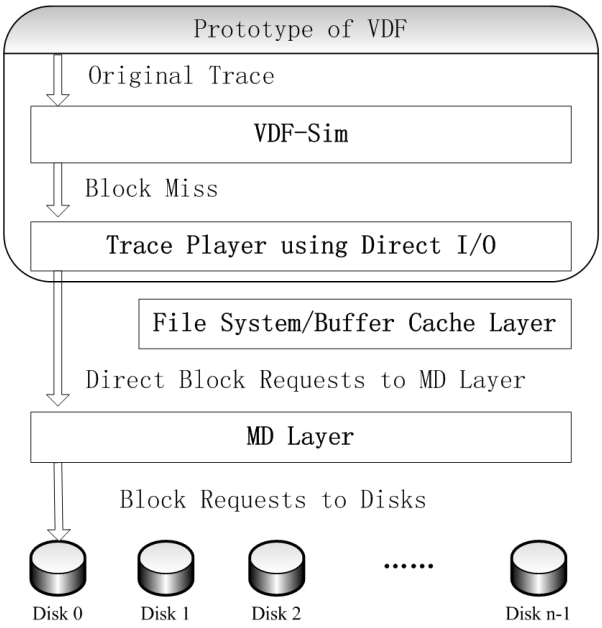


Figure 6: Architecture of VDF prototype.

6 Prototyping of VDF

To further evaluate VDF, we implemented a prototype of VDF in a software RAID system in Linux known as *MD*. In this section, we present our measurement results, including the efficiency of online recovery duration in full-bandwidth reconstruction mode and system service rate under the degraded mode without reconstruction.

6.1 Evaluation Methodology

Measurements on real world systems are welcome in research of computer systems. However, implementation in a real system is a lengthy process and always complex and challenging. Here, we use a straightforward and accurate measurement approach to evaluate the efficiency of VDF. The architecture of our prototype is shown in Figure 6. First, we collect the cache miss information during our simulation in Section 5, which includes not only the block ID but also the real access timestamps. Then, we treat the RAID as a file device, and use an application in user mode to play the traces we have collected from our simulations which is similar to RAIDmeter [1, 11]. However, the difference is that our application uses direct I/O (available in Linux 2.6 and up) instead of buffered I/O to avoid the requests being recached by the file system buffer cache. All missed I/O requests sent to the MD layer directly. Thus our simulation and the application join together to exploit the buffer cache and replacement algorithm. The trace player is also written in C and the source code is approximately 500 lines.

In our experiment, we evaluated the effectiveness of VDF, including the online reconstruction duration in full-bandwidth reconstruction mode, and the system serviceability in the degraded mode without reconstruction. For online reconstruction in full-bandwidth reconstruction mode, an open-loop measurement approach is adopted, where all filtered traces are played according to their timestamps as recorded in the original trace file. For degraded mode, a closed-loop measurement approach is adopted, where all filtered traces are played only according to their original sequence one by one and without any interval, to find the system serviceability.

6.2 Experimental Environment

In our experiment, we employ a SuperMicro storage server with two Intel(R) Xeon(R) X5560 @2.67GHz (six cores) CPUs, 12GB DDR3 main memory. All disks are Western Digital WD10EALS Caviar Blue SATA2, which are connected by an Adaptec 31605 SAS/SATA RAID controller with a 256MB dedicated cache. We disabled the RAID function of the controller and only used the direct I/O mode to connect each disk. The operating system is Linux Fedora 12 x86_64 with the kernel version of 2.6.32.

In Linux, there is a software implementation of RAID called Multiple Devices *MD*, which is popular in verification of RAID optimization scheme [1, 11]. To facilitate the analysis and verification of VDF cache, we also used MD as our experimental platform. We used the default settings of MD: the chunk size is 64KB, the number of stripe-heads is 256 and the data layout is left-symmetric. In our open-loop testing, the minimum

Table 3: Experimental Results of an Open-loop Testing Using 5 to 8 Disks

Disks	Blocks	LRU (s)	VDF-LRU (s)	Improvement	LFU (s)	VDF-LFU (s)	Improvement
5 disks	131072	2662	2543	4.5%	2710	1929	28.8%
	262144	2958	1935	34.6%	2851	1531	46.3%
	524288	1845	1407	23.7%	1786	1310	26.7%
6 disks	131072	1176	1147	2.5%	1175	964	18.0%
	262144	1234	943	23.6%	1226	921	24.9%
	524288	1027	818	20.4%	1005	806	19.8%
7 disks	131072	730	685	6.2%	733	652	11.1%
	262144	758	659	13.1%	761	657	13.7%
	524288	691	599	13.3%	687	598	13.0%
8 disks	131072	504	485	3.8%	509	485	4.7%
	262144	558	501	10.2%	560	501	10.5%
	524288	527	483	8.4%	526	479	8.9%

reconstruction bandwidth is set to 100MBps to utilize all remaining bandwidth for reconstruction besides the bandwidth taken by user workloads.

As VDF targets the storage server consisting of disk arrays which run under faulty conditions, we chose the server-end trace SPC-1-web as our experimental material, which spans a 60GB dataset. The workload is filtered by 131,072 to 524,288 blocks in our simulation to generate the experimental inputs. In the open-loop measurement, we test VDF with 5 to 8 disks. The results are reported in terms of reconstruction speed. The improvements of VDF over the original LRU and LFU are calculated. In the close-loop measurement, we used a multi-threaded application to play the filtered trace to measure the service rate. We also evaluated the impact of thread number to service rate, in addition to the impact of the number of blocks and the number of disks. The results are reported as system service rate improvement by VDF cache compared to original LFU and LRU. We ran each test three times and report the average. The results are very stable and consistent as the difference among all three rounds was very small (less than 5%).

6.3 Open-loop Measurement Results and Analysis

Table 3 describes the results under an open-loop testing using the SPC-1-web trace, where the number of disks ranges from 5 to 8 and the number of blocks ranges from 131,072 to 524,288. The experimental results of the open-loop testing show that compared to the original LRU and LFU algorithms, the VDF-optimized algorithm speeds up the online reconstruction process. The speedup of VDF-LFU is up to 46.3% compared to LFU. VDF-LRU speeds up the online reconstruction by up to 34.6% compared to LRU.

With the same number of cache-resident blocks, the experimental results show that the improvement of reconstruction durations of VDF-LFU to LFU decreases with the increased number of disks. A similar trend is also observed for the improvement of VDF-LRU over LRU, except for a smaller improvement of RGR using VDF when the number of blocks is 131,072. These trends are in contrast to our previous simulation results where the efficiency of VDF tends to be more sufficient with increased number of disks. From Equation 3, the improvement of reconstruction durations of VDF-LFU to LFU, which is presented by $Imprv$, could be described with Equation 4. $BW_U \times RGR_{VDF}$ is always less than BW otherwise the system would be overloaded, so $\frac{BW}{RGR_{VDF}}$ is larger than BW_U. VDF works in most cases where $\frac{RGR_{ORI}}{RGR_{VDF}}$ is larger than 1, thus $\frac{BW_U * RGR_{ORI}}{RGR_{VDF}}$ is larger than BW_U. Therefore, the change rate of improvement according with the number of total disks should only be determined by the changing rates of $BW_U \times RGR_{ORI}$ and BW. Obviously, BW linearly grows with the total number of disks. From the simulation result, we can find that the changing rate of $BW_U \times RGR_{ORI}$ is slower than that of BW with the number of disks from 5 to 8. Therefore, in most of the above cases, the improvement of reconstruction durations of VDF cache to original cache decreases with the increased number of disks.

$$RD_{Imprv} = \frac{\frac{BW_U * RGR_{ORI}}{RGR_{VDF}} - BW_U}{\frac{BW}{RGR_{VDF}} - BW_U} \qquad (4)$$

With the same number of disks, we notice that the reconstruction duration of the trace filtered by 131,072 blocks is less than the reconstruction duration of trace filtered by 262,144 blocks in many cases. On one hand, as we use a number of blocks to warm up the cache, this part of the miss information is not recorded in our filtered trace file. The first 131,072 block misses in the

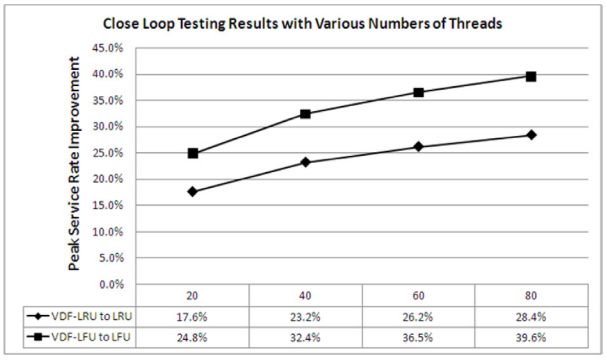


Figure 7: Service rate improvement of VDF in degraded mode of a RAID-5 of 8 disks. The number of blocks is 524,288 and the number of threads ranges from 20 to 80.

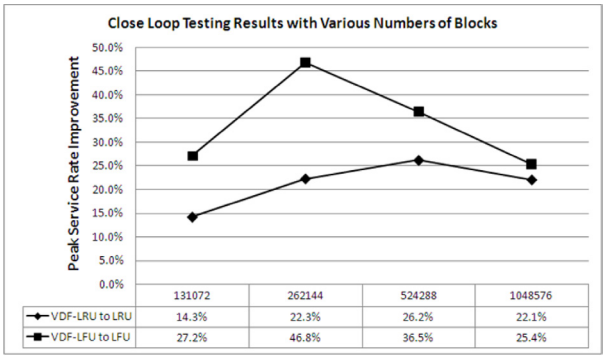


Figure 8: Service rate improvement of VDF in degraded mode of a RAID-5 of 8 disks. The number of blocks ranges from 131,072 to 524,288 and the number of threads is 60.

trace filtered by 262,144 blocks has a lower average arrival rate than the remaining part. On the other hand, when the number of blocks in the cache is 131,072 or 262,144, the cache is too small to find the hot blocks, which implies fewer hits in those cases and the RGRs are similar. Therefore, the reconstruction duration of the trace filtered by 131,072 blocks might be less than that of the trace filtered by 262,144 blocks in those cases.

6.4 Close-loop Measurement Results and Analysis

Figures 7, 8, and 9 present the close-loop testing results under different scenarios with various numbers of threads, disks, and data blocks. The results are reported as service rate improvement, which is inversely proportional to the *Play Duration* (**PD**) of a whole filtered trace. For example, the corresponding service rate improvement of VDF-LRU to LRU should be calculated by $\frac{PD_{LRU}-PD_{VDF-LRU}}{PD_{VDF-LRU}}$.

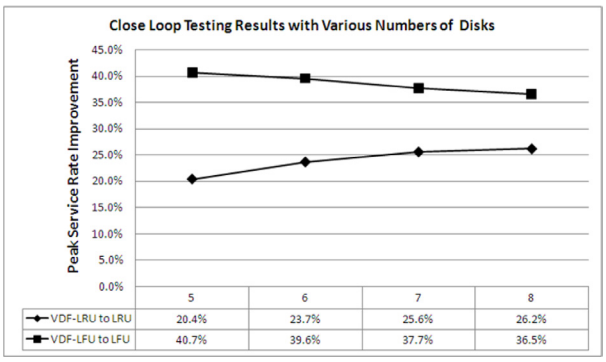


Figure 9: Service rate improvement of VDF in degraded mode of a RAID-5 of 5 to 8 disks. The number of blocks is 524,288 and the number of threads is 60.

From the experimental results, we find that VDF is effective in improving the system service rate. Compared to LFU, VDF-LFU improves the system service rate up to 46.8% with 60 threads under 8 disks and 262,144 blocks. Compared to LRU, VDF-LRU improves the system service rate by up to 28.4% with 80 threads under 8 disks and 524,288 blocks.

With the increasing number of I/O threads, the service rate improvement increases accordingly and gets close to the theoretical value calculated by the simulation results. Although the whole trace would be evenly distributed on all disks due to the round-robin addressing in RAID, the incoming user requests might not be evenly distributed on all the disks during a short period. Therefore, with a larger number of threads, which implies a longer scheduling window, the distribution of incoming user requests is more balanced and the user service rate is closer to the maximum system service rate.

With the same number of blocks and a fixed number of threads, the service rate improvement of VDF-LRU to LRU is consistent with the trend of the simulation result. However, to our surprise, the results of VDF-LFU to LFU were just opposite with the trend of the simulation result. As per our analysis, this was primarily due to two reasons. First, from the simulation result, the relative RGR reduction of VDF-LFU to LFU with 524,288 blocks is in a small area from 33.7% to 35.6%. Second, the number of concurrent threads is fixed, which means that the number of threads per disk would increase with the decreased total number of disks. Thus, based on the above analysis, when the total number of disks is small, the improvement is closer to the theoretical value. Therefore, under close to theoretical peak service rates and more I/O threads per disk, the trend of service rate improvement of VDF-LFU to LFU is very possibly opposite with the trend of the simulation result. As a result, with the same number of disks and a fixed number

of threads, which means a fixed number of I/O threads per disk, the service rate improvement is quite consistent with the trend of the simulation results.

6.5 Further Discussion

Several more issues deserve further discussion. The first issue is the implementation cost of VDF. As we mentioned in Section 4, to make the smooth conversion between the original cache algorithms and the VDF-based algorithms, two types of stacks should be employed to implement VDF cache. This adds both spatial and temporal overhead. The spatial overhead include the extra information in each block head such as the timestamp and the extra stack pointer of the local stack. Compared to the buffer cache size, this overhead is very small. The temporal overhead is the computation of the weight of block at the bottom of each LS stack. Due to the high computation ability of today's CPU, this should not influence the overall system performance.

Second, can we integrate VDF into other optimizations in faulty mode? As the VDF cache essentially reduces the user requests to the surviving disks, it can be integrated with other optimizations in faulty mode, such as optimization on data/parity/spare layout and reconstruction workloads. The approach of redirection of reads utilizes the reconstructed data in a spare disk to serve part of the reads to the faulty disk. Thus the miss penalty of these reconstructed data block is zero in terms of extra requests to the surviving disks. There still exist hot data with large miss penalty on faulty disks. Therefore, VDF can still help.

Third, could RGR be suitable to describe the status of write operation? From its definition, RGR is determined by MP_i and p_i. The calculation of p_i in write operations is similar to read operations. However, the calculation of MP_i in write operations is quite different from read operations, as they might be done with two approaches based on the parity distribution in RAID with faulty disk(s). One is the Read-Modify-Write, and the other is Parity-Reconstruction-Write. Here, we take an example of short writes on an n-disk RAID-5 with one faulty disk to demonstrate the MP_i calculation for write operations. Once a short write is sent to the surviving disks and the corresponding parity is not on the faulty disk, the Read-Modify-Write should be performed which results in two reads and two writes on the surviving disks, and thus the MP_i is 4. Otherwise, the Parity-Reconstruction-Write should be performed which results in $n-1$ reads and one write on the surviving disks, and thus the MP_i is n.

7 Conclusions and Future Work

In this paper, we present an asymmetric buffer cache replacement strategy, named Victim (or faulty) Disk(s) First (VDF) cache, to improve the reliability and performance of a RAID-based storage system, particularly under faulty conditions. The basic idea of VDF is to treat the faulty disks more favorably, or give a higher priority to cache the data associated with the faulty disks. The benefit of this scheme is to reduce number of the cache miss directed to the faulty disk, and thus to reduce the I/O requests to the surviving disks overall. Less disk I/O activity caused by the user workflow will (1) improve the performance of the disk array, and (2) allow more bandwidth for online reconstruction which in turn speeds up the recovery, and thus improves the reliability. Our results based on both simulation and prototyping implementation has demonstrated the effectiveness of VDF in terms of reduced disk I/O activities and a faster recovery.

To further understand VDF, we have the following plans as our future work. First, we plan to build VDF into more general cache algorithms such as CLOCK [30] and ARC [26]. Second, we are working to implement VDF in the kernel level and thus to directly run real benchmarks to conduct more extensive measurements. Third, in addition to RAID-5, we will investigate the scheme to apply VDF to other RAID levels such as RAID-4 and RAID-6.

Acknowledgments

We are very grateful to our shepherd Erez Zadok and anonymous reviewers for their helpful comments. This research is sponsored in part by the National Basic Research 973 Program of China under Grant No. 2011CB302303, the National Natural Science Foundation of China under Grant No. 60933002, the National 863 Program of China under Grant No. 2009AA01A402, and the Innovative Foundation of Wuhan National Laboratory for Optoelectronics. The author He's work is supported by the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1102605, CCF-1102624, and CNS-1102629. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] S. Wu, H. Jiang, D. Feng, L. Tian, and Bo Mao. WorkOut: I/O workload outsourcing for boosting RAID reconstruction performance. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, San Francisco, USA, February 2009.

[2] D. A. Patterson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the*

1988 ACM SIGMOD international conference, Chicago, Illinois, USA, June 1988.

[3] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems*, pages 23–35, Boston, Massachusetts, USA, October 1992.

[4] J. Menon and D. Mattson. Distributed sparing in disk arrays. In *Proceedings of the 37th international conference on COMPCON*, pages 410–421, San Francisco, California, USA, Feb 1992.

[5] Q. Xin, P. L. Miller, and T. J. E. Schwarz. Evaluation of distributed recovery in large-scale storage systems. In *Proceedings of 13th IEEE International Symposium on High Performance Distributed Computing*, pages 172–181, June 2004.

[6] G. K.M, X. Li, and J. J. Wylie. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. In *IEEE 26th Symposium on Mass Storage Systems and Technologies*, Incline Village, NV, May 2010.

[7] S. Wan, Q. Cao, C. S. Xie, B. Eckart, and X. He. Code-M: A non-MDS erasure code scheme to support fast recovery from up to two-disk failures in storage systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, IL, USA, June 2010.

[8] R. Y. Hou, J. Meno, and Y. N. Patt. Balancing I/O response time and disk rebuild time in a RAID5 disk array. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, pages 70–79, Jan 1993.

[9] J. Y.B. Lee and J. C.S. Lui. Automatic recovery from disk failure in continuous-media servers. *The Computer Journal*, 13(5):499–515, May 2002.

[10] M. Holland, G. A. Gibson, and D. P. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 422–431, Toulouse , France, Jun 1993.

[11] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 277–290, February 2007.

[12] L. Tian, H. Jiang, and D. Feng. Implementation and evaluation of a popularity-based reconstruction optimization algorithm in availability-oriented disk arrays. In *24th IEEE Conference on Mass Storage Systems and Technologies*, pages 233–238, San Diego, CA, USA, Sep 2007.

[13] R. R. Muntz and J. C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Databases*, pages 162–173, 1990.

[14] T. Xie and H. Wang. MICRO: A multilevel caching-based reconstruction optimization for mobile storage systems. *IEEE Transactions on Computers*, 57(10):1386–1398, Oct 2008.

[15] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 143–152, Boulder, Colorado, USA, May 1990.

[16] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, Boulder, Colorado, USA, May 1990.

[17] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD international Conference on Management of data*, pages 297–306, Washington, D.C., USA, May 1993.

[18] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of ACM*, 46(1):92–112, Jan 1999.

[19] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago de Chile, Chile, USA, September 1994.

[20] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–143, Atlanta, Georgia, USA, May 1994.

[21] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, Dec 2001.

[22] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings 2001 Annual USENIX Technical Conference*, pages 91–104, Boston, Massachusetts, USA, June 2001.

[23] J. F. Philbin Y. Zhou and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, June 2004.

[24] S. Jiang and X. Zhang. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 31–42, Marina Del Rey, California, USA, June 2002.

[25] S. Jiang and X. Zhang. Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance. *IEEE Transactions on Computers*, 54(8):939–952, Aug 2005.

[26] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, San Francisco, CA, USA, Mar 2003.

[27] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pages 14–16, San Francisco, CA, USA, December 2005.

[28] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of the USENIX Annual Technical Conference*, Santa Clara, CA, USA, Jun 2007.

[29] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *IEEE International Symposium on Workload Characterization*, Seattle, WA, USA, Sept 2008.

[30] F. J. Corbato. A paging experiment with the Multics system. *In MIT Project MAC Report MAC-M-384*, May 1968.

The Design and Evolution of Live Storage Migration in VMware ESX

Ali Mashtizadeh *Emré Celebi* *Tal Garfinkel* *Min Cai*
{*ali, emre, talg, mcai*}*@vmware.com*
VMware, Inc.

Abstract

Live migration enables a running virtual machine to move between two physical hosts with no perceptible interruption in service. This allows customers to avoid costly downtimes associated with hardware maintenance and upgrades, and facilitates automated load-balancing. Consequently, it has become a critical feature of enterprise class virtual infrastructure.

In the past, live migration only moved the memory and device state of a VM, limiting migration to hosts with identical shared storage. Live storage migration overcomes this limitation by enabling the movement of virtual disks across storage elements, thus enabling greater VM mobility, zero downtime maintenance and upgrades of storage elements, and automatic storage load-balancing.

We describe the evolution of live storage migration in VMware ESX through three separate architectures, and explore the performance, complexity and functionality trade-offs of each.

1 Introduction

Live virtual machine migration is a key feature of enterprise virtual infrastructure, allowing maintenance and upgrades of physical hosts without service interruption and enabling manual and automated load-balancing [1].

Live migration works by copying the memory and device state of a VM from one host to another with negligible VM downtime [2]. The basic approach is as follows: we begin by copying most of the VM state from the source host to the destination host. The VM continues to run on the source, and the changes it makes are reflected to the destination , at some point the source and destination converge – generally because the source VM is momentarily suspended allowing the remaining differences to be copied to the destination. Finally, the source VM is killed, and the replica on the destination is made live.

Earlier live migration solutions did not migrate virtual disks, instead requiring that virtual disks reside on the same shared volume accessible by both the source and destination hosts. To overcome this limitation, various software and hardware solutions to enable live migrations to span volumes or distance have been developed. One such solution is live storage migration in VMware ESX.

Live storage migration has several important use cases. First, zero downtime maintenance – allowing customers to move on and off storage volumes, upgrade storage arrays, perform file-system upgrades, and service hardware. Next, manual and automatic storage load-balancing – customers in the field already manually load balance their ESX clusters to improve storage performance and automatic storage load balancing will be a major feature of the next release of the VMware vSphere platform. Finally, live storage migration increases VM mobility in that VMs are no longer pinned to the storage array they are instantiated on.

Multiple approaches to live storage migration are possible, each offering different trade-offs by way of functionality, implementation complexity and performance. We present our experience with three different approaches: Snapshotting (in ESX 3.5), Dirty Block Tracking (in ESX 4.0/4.1) and IO Mirroring (in ESX 5.0). We evaluate each approach using the following criteria:

- **Migration time:** Total migration time should be minimized, and the algorithm should guarantee convergence in the sense that the source and destination copies of the virtual disk eventually match. We show that some algorithms do not guarantee convergence and carry the risk of not completing without significant disruption to the workload. We also emphasize predictability – Live storage migrations can take a while; predictability allows end users to better plan maintenance schedules.

- **Guest Penalty:** Guest penalty is measured in application downtime and IOPS penalty on the guest workload. All live migration technologies strive to achieve zero downtime – in the sense that there is no perceptible service disruption. However, live migration of any sort always requires some downtime during the hand-off from the source to the destination machine. Most applications can handle several seconds of downtime without any network connectivity loss. Highly available applications may only handle one or two seconds of disruption before an instance is assumed down. The final approach discussed in this paper exhibits no visible downtime and a moderate performance penalty.

- **Atomicity:** The algorithm should guarantee an atomic switchover between the source and destination volumes. This increases reliability and avoids creating a dependence on multiple volumes. Atomic switchover is a requirement to make physically longer distance migrations safe, and for mission critical workloads that cannot tolerate any noticeable downtime.

We also compare how the three approaches perform when migrating between volumes with similar and differing performance, and analyze their performance with a synthetic *online transaction processing* (OLTP) and real application (Exchange 2010) workload.

2 Design

We compare three approaches to live storage migration. The first, based on snapshots was introduced in ESX 3.5, the second based on an iterative copy with a Dirty Block Tracking (DBT) mechanism was introduced in ESX 4.0 and refined in 4.1, and the most recent approach leveraging synchronous IO Mirroring, will ship with ESX 5.0.

2.1 Background

Live storage migration can take place between any two storage elements whether over fiber channel, iSCSI or NFS.

All approaches to live migration follow a similar pattern. A virtual disk(s) is migrated (copied) from a source volume to a destination volume. Initially, a running virtual machine is using the virtual disk on the source volume. As the disk on the source is copied to the destination, bits are still being modified on the source copy. These changes are reflected to the destination so that source and destination ultimately converge. Once the two copies are identical, the running VM can be retargeted to use the destination copy of the virtual disk.

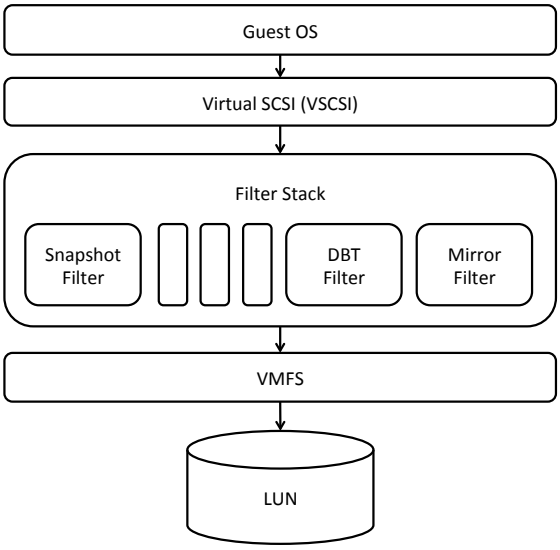


Figure 1: Simplified ESX storage architecture diagram: The guest operating system issues IO through the virtualized SCSI (VSCSI) storage stack. The IOs are passed through a stack of one or more filter drivers. As an example, this diagram shows the snapshot filter that implements the virtual disk snapshot file format. Once IOs pass through the filter stack, they are translated into file handles that are used by VMware's VMFS file-system or NFS client service.

Our migration system is built with a combination of a storage stack filter driver and a user-level thread. The user-level thread, which facilitates the migration, is a part of the VM management executive (VMX). The filter drivers are depicted in the ESX storage stack in Figure 1.

More recent architectures i.e. DBT and IO Mirroring, use the data mover (DM) copy engine that was added in ESX 4.0. The DM is a kernel service that copies disk blocks between locations with only DMAs. This eliminates user-space and kernel crossing overheads, and enables the use of copy off-load engines sometimes present in storage arrays [3].

2.2 Snapshotting

Snapshotting, our first version of live storage migration, was built to enable VMFS file system upgrades. To upgrade from VMFS version 2 to version 3, version 2 volumes were rendered read-only and virtual disks migrated onto version 3 volumes.

Snapshotting leverages virtual machine snapshots, to recap how snapshots work: when a snapshot is taken, all disk contents at snapshot time are preserved. Future modifications to the disk are logged in a separate snapshot file. Multiple levels of snapshots are possible, and

multiple snapshots can be consolidated into a single disk or a snapshot by applying modifications in each to the previous snapshot or base disk. Once consolidated, intermediate snapshots can be discarded.

The migration begins by taking a snapshot of the base disk, all new writes are sent to this snapshot. Concurrently, we copy the base disk to the destination volume. Our first snapshot may reside on the source or destination volume, though the former is preferable to minimize the time the virtual disk spans two volumes.

After we finish copying the base disk, we take another snapshot. We then consolidate the first snapshot into the base disk. By the time this consolidation is complete, we expect more writes have occurred, the result is again a delta between our source and destination.

We repeat the process until the amount of data in the snapshot becomes smaller than a threshold. Once this threshold is reached, the VM is suspended, the final snapshot is consolidated into the destination disk, and the VM is resumed, now running with the virtual disk on the destination volume.

Snapshot consolidation cannot be done using an active writable snapshot due to the risk of inconsistency. Consequently, the VM must be suspended to render it inactive, resulting in downtime in our final consolidation step. Online consolidation of a read-only snapshot is possible, allowing us to implement our iterative consolidation step with minimal down time. A threshold, that can be determined dynamically, specifies when to perform the final consolidation and what the resulting downtime will be.

Snapshotting inherits the simplicity and robustness of the existing snapshot mechanism. When compared with the next design (DBT), Snapshotting shows significant resilience in the face of differing performance characteristics on the source and destination volumes. However, it also exhibits two major limitations.

First, migration using snapshots is not atomic. Consequently, canceling a migration in progress can leave the migration in an intermediate state where multiple snapshots and virtual disks are spread on both source and destination volumes. Similarly, a storage failure on either volume necessitates termination of the VM. Snapshotting is not suitable for long distance migrations to a remote destination volume, since a network outage can cause an IO stall requiring us to halt the VM. We attempt to create the initial snapshot on the source volume to help mitigate this issue.

Second, there are performance and space costs associated with running a VM with several levels of snapshots. More specifically, when iteratively consolidating snapshots there are multiple outstanding snapshots, a writable snapshot that is absorbing all new disk modifications and a read-only snapshot that is being consolidated. Using both snapshots concurrently increases memory and IO overheads during the migration. In the worst case, assuming both levels of snapshots grow to the size of the full disk, the VM may temporarily use three times its normal disk space.

2.3 Dirty Block Tracking

Our next design sought to overcome the limitations of Snapshotting, including downtime penalties from consolidation overhead and the lack of atomic switches from source to destination volumes for failure robustness.

Our approach, informed by our experience with live VM migration, uses a very similar architecture. *Dirty Block Tracking* (DBT) uses a bitmap to track modified aka dirty blocks on the source disk, and iteratively copy those blocks to the destination disk.

With DBT, we begin by copying the disk to the destination volume, while concurrently tracking dirty blocks on the source disk in the DBT kernel filter. At the end of the first copy iteration, we atomically get and clear the bitmap from the kernel filter, blocks marked in the bitmap are copied to the destination. This process is repeated until the number of dirty blocks remaining at each cycle stabilizes i.e. no forward progress is being made or a threshold based on a target downtime is reached. At this point, the VM is suspended and the remaining dirty blocks are copied.

DBT is done concurrently with bulk copying the disk contents to the destination. If a block is dirtied but not yet copied, we do not need to track that block, as it will later be bulk copied. Using this technique results in a roughly 50% speedup for the first copy iteration, assuming a workload consisting of uniformly distributed random writes, leading to an optimization we call *incremental DBT*.

DBT has several attractive properties. Operating at the block instead of snapshot granularity makes new optimizations possible. Also, DBT guarantees atomic switch-over between the source and destination volumes i.e. a VM on the source can continue running even if the destination hardware or link fail, improving reliability and making DBT suitable for migrating in less reliable conditions, such as over the WAN.

DBT also introduces new challenges. Migrations may take longer to converge if the guest workload is write intensive. If the workload on the source dirties blocks at a rate greater than the copy throughput then the migration cannot converge. The only remedy is to quiesce the guest, imposing significant downtime, or to cancel the migration.

2.3.1 Hot Block Avoidance

We present an optimization that detects frequently written blocks and defers copying them. We discuss the motivations for this optimization in section 2.3.2. In Sections 2.3.3 and 2.3.4 we present the implementation and some preliminary results. Finally, in Sections 2.3.5 we explore some of the challenges we encountered implementing this solution. Due to these challenges, this feature was never enabled by default in a shipping release. While we present these optimizations in the context of DBT, we hope to apply them to future versions of IO Mirroring.

2.3.2 Distribution of Disk IO Repetition

Real-world disk IO often exhibits temporal and spatial locality. To help us better understand locality in a common enterprise workload, we analyzed VSCSI traces from an Exchange 2003 workload with 100 users.

Our workload was generated using the Exchange Load Generator [4] in a VM configured with three disks: a system disk of 12GB with Windows 2003 server, a mailbox disk of 20GB, and a log disk of 10GB. Exchange is configured to use circular logs.

Our results are shown in Figure 2 that plots logical block numbers (LBNs) sorted by decreasing popularity i.e. most to least frequently written blocks, for all three disks. All traces follow a zipf-like distribution. Once hot blocks are identified, we can ignore them during the iterative copy phase, and defer copying of hot blocks to the end of the migration. This eliminates numerous repeated copies that reduces IO costs and overall migration time.

2.3.3 Multi-stage Filter for Hot Blocks

We collect data from the DBT filter to identify candidate hot blocks. Using a hash table to index the repetition counters for large disks would be quite memory intensive. Therefore, we use a multi-stage filter [5] to identify blocks that have been written at least t times, where t is a threshold. The multi-stage filter is similar to a counting version of bloom filter, which can accurately estimate the dirty blocks. Multi-stage filters provide a compact representation of this data.

Our multi-stage filter has n stages. Each stage includes an array of m counters and a uniform hash function H_i, where $0 \leq i \leq n$. When a block with LBN x gets modified, n different hash values of a block are calculated, and the corresponding counters in all stages are increased. The hotness of a block can be determined by checking the counters of the block in all stages. When all counters are greater than a threshold t, the corresponding block is considered to be hot. Since the collision probability of all n counters decrease exponentially with n, the multi-stage filter is able to filter out hot blocks accurately with limited memory.

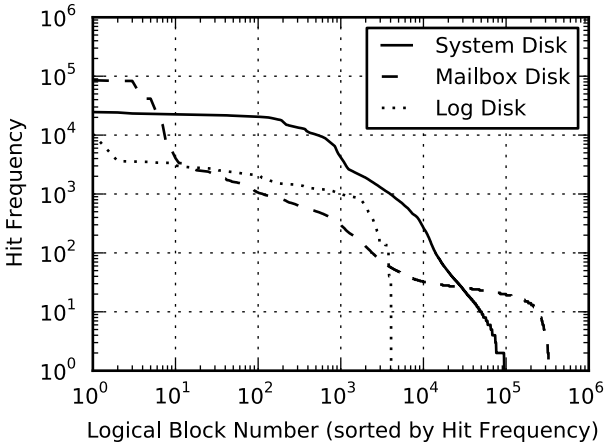


Figure 2: Distribution of Disk Write Repetition of Exchange Server Workload. The x-axis shows the logical block number (LBN) of the disk sorted by hit frequency. The y-axis shows the hit frequency for disk blocks from hottest to coldest. The writes follow a zipf-like distribution.

2.3.4 Analysis of Hot Block Avoidance

Our hot block avoidance algorithm uses the heat map from our multi-stage filter to determine which blocks are hot. Sampling is done during the initial copy phase. In the iterative copy phase, we query the multi-stage filter and defer copying of the hot blocks. At the end of the migration, hot blocks are copied, prior to the last copy iteration.

To appreciate the potential benefits, consider the distribution of write frequencies for the Exchange workload shown in Figure 2. Several hundred megabytes of blocks are hot, ignoring these until the final copy iteration can yield substantial benefits.

These benefits can be seen in Figure 3, a migration using same Exchange workload described previously, with and without our hot block and incremental DBT optimizations. The initial copy phase is not shown, as it is independent of optimizations. Shorter bars on the left represent a migration with optimizations, the taller bars on the right, without. Iterations 5 through 10 are not present for our optimized case, since incremental DBT and hot block avoidance eliminates the need for those iterations.

The consistent height of the bars with the red hatch pattern shows the hot block avoidance algorithm detected the approximate working set correctly. Note that the blocks labeled with the red hatch pattern are *not copied*

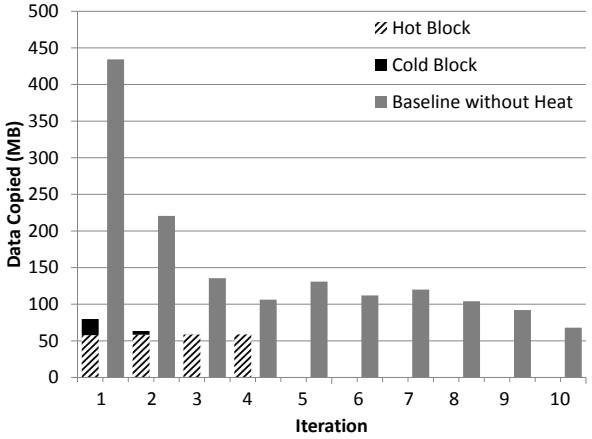


Figure 3: Dirty blocks copied vs. Iteration number. Exchange workload migration with and without our hot block and incremental DBT optimizations. Shorter bars on the left represent a migration with optimizations, the taller bars on the right, without. The initial copy phase is not shown, as it is independent of optimizations. Iterations 5 through 10 are not present for our optimized case, since hot block avoidance eliminates the need for those iterations.

for the first two iterations. We displayed the graph like this to make it easier to see the remaining blocks that need to be copied. Most of the blocks copied in third iteration are so hot that they are necessarily copied again during the switchover and completion of the virtual machine. In our implementation we attempted to copy the hot blocks in next to last iteration in order to reduce the remaining blocks for the final iteration. This is critical because the final iteration occurs while the VM execution is suspended and accounts for our downtime.

Incremental DBT saves more than 50% on the first iteration. Our workload issues more writes towards the end of the disk, and incremental DBT allows us to ignore those blocks during our iterative copy phase, allowing them to be taken care of by the bulk copy of the base disk that is happening concurrently.

2.3.5 Problems with Hot Block Avoidance

Our hot block avoidance algorithm performed well in most scenarios, however, we encountered several problems. First, we found hot block avoidance complex to tune. Success was hard to reason about and we were concerned about harming the performance of untested workloads.

Hot block avoidance also consumed significant amounts of memory. Even with the multi-stage filter, we could envision large VMs consuming upwards of a gigabyte of memory. Our customers, many of whom run with memory overcommitted, could find additional memory pressure problematic.

Finally, some workloads e.g. the OLTP workload discussed in our evaluation, have little or no temporal locality, and thus receive minimal benefit from this optimization.

2.4 IO Mirroring

DBT is an adaptation of the technique used for live virtual machine migration, namely, iterative pre-copying of virtual machine memory pages. While DBT has benefits over Snapshotting, they come at the cost of complexity. To improve on DBT, we note a critical distinction between virtual memory and storage systems.

Virtual machine memory accesses are usually transparent to the hypervisor and write traps are quite expensive. Consequently, write traps are used only to note if an already copied page has again been dirtied – i.e. only the first write to a copied page is trapped – necessitating an iterative copying approach where all the writes to a page of a given "generation" are captured by copying the entire dirty page. In contrast, intercepting all storage writes is relatively cheap. Our next approach leverages this observation, using a much simpler architecture based on synchronous write mirroring.

IO Mirroring, our most recent architecture, works by mirroring all new writes from the source to the destination concurrent with a bulk copy of the base disk. We again use a filter driver as shown in Figure 1. Our bulk copy process is implemented using the VMKernel data mover (DM). We drive the copy process from user-level by issuing DM operations. The DM issues reads and writes directly to the underlying file without the intervention of the filter driver. Thus, if a VM could issue a write while a DM read operation is in progress, without a synchronization mechanism we would copy an outdated version of the disk block. To prevent this situation, the filter driver implements a synchronization mechanism to prevent DM and VM IOs to the same region. When a DM operation is issued first the filter acquires a lock on the region in question, and then releases it on completion.

Locking in the IO Mirroring filter driver works by classifying all VM writes into one of three types: writes to a region that has been copied by the DM, writes to a region being copied by the DM, and writes to a region that will be copied by the DM. Two integers are used to maintain the disk offsets that delineate these three regions.

Writes to a region that has already been copied will be mirrored to the source and destination – as the DM has already passed this area and any new updates must be reflected by the IO Mirror. Writes to the region currently

being copied (in between the two offsets) will be deferred and placed into a queue. Once the DM IO completes we enqueue those writes and unlock the region by updating the offsets. As part of the updating operation we wait for inflight writes to complete. The final region is not mirrored and all writes are issued to the source only – as eventually any changes to this area will be copied over by the DM. Reads are only issued to the source disk.

IO Mirroring fulfills all of our criteria, guaranteeing an atomic switchover between the source and destination volumes, making this method viable for long distance migrations. It also guarantees convergence as the mirrored VM IOs are naturally slowed down to the speed of the slower volume.

2.5 Implementation Experience

Snapshotting benefited from leveraging the existing snapshot mechanism, making it simpler to implement, and easier to bring into a hardened production quality state. Further, it was the only approach that was feasible for the file system upgrade use case that originally motivated its creation. For this use case, the source virtual disk lives on an older read-only file system, and both DBT and IO Mirroring require a writable source disk. Finally, it required no complex tuning. Unfortunately, it also inherited substantial limitations from leveraging snapshots, most notably the atomicity, and the performance limitations of snapshots.

DBT overcame most of those performance inadequacies, but introduced many parameters that required significant engineering effort to tune. Specifically, for the convergence detection logic that determines whether a migration needs additional copy iterations, is safe to complete, or needs to be terminated.

This logic required several threshold values to determine whether the remaining dirty blocks at the end of the iteration seem to be getting smaller. If the algorithm detects a significant reduction in the remaining dirty blocks, it continues for another iteration. If there is no discernible reduction or possibly an increase, the algorithm determines whether to complete or abort the migration.

Two scenarios occur often enough that may cause a noticeable increase in the dirty blocks remaining. First, the workload may make a burst of changes e.g. a database may flush its buffer cache, causing the migration progress to temporarily regress. To handle this, the algorithm monitors progress for the last two copy iterations. Analyzing the last two iterations prevents nearly all migration aborts due to workload spikes. The second cause of failure to converge is a slow destination. If the workload running in a VM is too fast for the destination the migration will terminate. There is no solution other than to ask the user to quiesce such workloads manually.

IO Mirroring removed all of the tunable parameters and convergence logic. Using a synchronous mirror naturally throttles the workload to the speed of the destination volume. Switching to this approach eliminated significant engineering and performance testing effort. To our surprise, customers seemed most interested in the predictability aspect of IO Mirroring, as it allows them to better plan their maintenance schedules.

3 Evaluation

We evaluated total migration time, downtime, guest performance penalty and convergence, using synthetic (Iometer [6]) and real application (Exchange 2010) workloads for each of our architectures. We also present the IOPS profile for each architecture over the duration of a migration.

Our synthetic workload uses Iometer in a VM running Windows Server 2003 Enterprise, we varied disk size and outstanding IOs (OIOs) to simulate workloads of varying size and intensity. The IO pattern simulates an OLTP workload with 30% write, 70% read of 8KB IO commands with a 32GB preallocated virtual disk. We used Exchange 2010 for our application workload with loads of 44 tasks/sec and 22 tasks/sec.

Snapshotting and DBT were evaluated using ESX 4.1. IO Mirroring was evaluated using a pre-release version of ESX 5.0. Our snapshot implementation was first available in ESX version 3.5 however, we used the version in ESX 4.1 that included support for the DM and other major performance improvements to get a more fair comparison with DBT.

Our synthetic workload ran on a Dell Poweredge R710 server with dual Intel Xeon X5570 2.93 GHz processors, and two EMC CX4-120 arrays connected to the server via 8Gb Fibre Channel(FC) links. We created 450GB sized VMFS version 3 volumes on each array. Our test VM has a 6GB system disk running Windows Server 2003 and Iometer, and a separate data disk. The Snapshotting implementation requires all disks to move together to the same destination. For a fair comparison we migrated the system and data disk for all architectures.

Our application workload ran on a Dell PE R910 4 socket 8-core Intel Nehalem-EX processor with 256GB of memory. Migration is done with 6 disks in RAID-0 configuration on the same EMC CX3-40 Clariion array with separate spindles. Our Exchange 2010 VM is configured with 8-vCPU with 28GB of memory and contains multiple virtual disks. We only migrated the 350GB Mailbox disk containing 2000 user mailboxes. We omitted Snapshotting from this workload because it requires all disks to be migrated together.

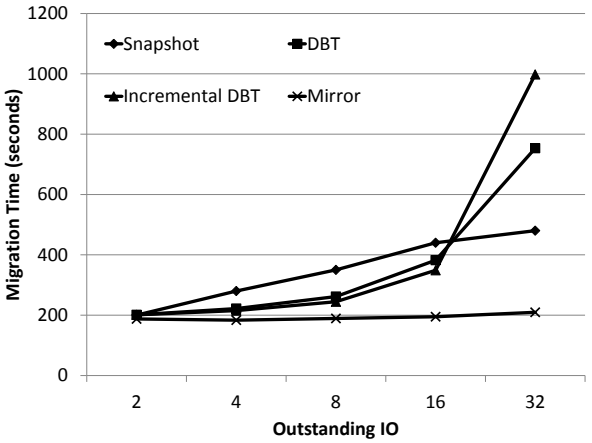


Figure 4: Migration Time vs. OIO. IO Mirroring exhibits the smallest increase by 11.8% at most. DBT variants exhibit the largest increase up to 2.74x and 3.96x. At 32 OIOs, the DM struggles to keep up with the workload's dirty rate. Snapshotting exhibits a 1.4x increase.

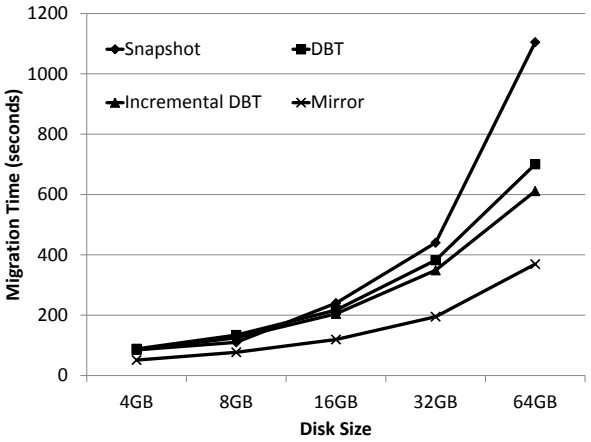


Figure 5: Migration Time vs. Disk Size. The x-axis denotes only the data disk size. Migration time includes the additional 6GB system disk. The migration times of IO Mirroring grows less than 3.4% with increasing disk size. DBT, Incremental DBT and Snapshotting takes 13%, 2.4% and 85% longer than expected.

3.1 Migration Time

Minimizing total migration time reduces the impact on guest workloads, decreases response time for maintenance events, and makes load balancing more responsive. Ideally, migration time should not vary when workload size and intensity change, as this makes migrations more predictable, easing manual and automated planning.

Total migration time vs. OIOs for our synthetic workload is shown in Figure 4. For OIOs less than 16, each architecture performs better than the previous one. Incremental DBT does marginally better than DBT, because the incremental dirty block tracking improvement reduces the number of dirty blocks copied in the first iteration. The VMKernel data mover (DM), used by both DBT variants, supports a maximum of 16 OIOs. Consequently, for workloads with more than 16 OIOs, Snapshotting outperforms both DBT variants, which has a bottleneck on the DM. IO Mirroring consistently offers the lowest total migration time.

IO Mirroring also offers the smallest change in migration time under increasing load, as we see in Figure 4. Migration time only grows by 11.8% when changing OIOs from 2 to 32, a 4.9x increase in guest write and read throughputs. In contrast, migration time increases by 1.4x for Snapshotting, and 2.74x and 3.96x for DBT and incremental DBT. IO Mirroring is less sensitive to OIOs because it implements a single pass copy operation. The increased IO slows that single pass copy rather than inducing additional copy iterations.

For comparison with the ideal case, we performed an off-line disk copy with a 32GB virtual disk and 6GB system disk, it took 176 seconds on the same hardware setup. Migration with IO Mirroring with 2 OIO and 32 OIO OLTP workloads took only 5.8% and 15.7% longer to complete.

Type	Migration Time	Downtime
DBT	2935.5s	13.297s
Incremental DBT	2638.9s	7.557s
IO Mirroring	1922.2s	0.220s
DBT (2x)	Failed	-
Incremental DBT (2x)	Failed	-
IO Mirroring (2x)	1824.3s	0.186s

Table 1: Migration time and downtime for DBT, Incremental DBT, and IO Mirroring with the Exchange workload. The double intensity version only completes with IO Mirroring.

Total migration time vs. disk size for our synthetic workload is shown in Figure 5. Again, each architecture migrates faster than the previous one. Generally migration time grows linearly with disk size however, for 64GB disks, Snapshotting performs worse than expected. This occurs because all subsequent snapshots grow in size leading to an increase in the number of snapshot creation and consolidation iterations. IO Mirroring exhibits minimal change as the disk size increases, with migration time growing less than 4%. Our figures include the 6GB system disk's migration time. Thus, the migration time for the 4GB and 64GB data disk tests corresponds to a seven fold increase.

Our Exchange workloads are depicted in Table 1. For the initial run, Incremental DBT offers a 11.2% reduction

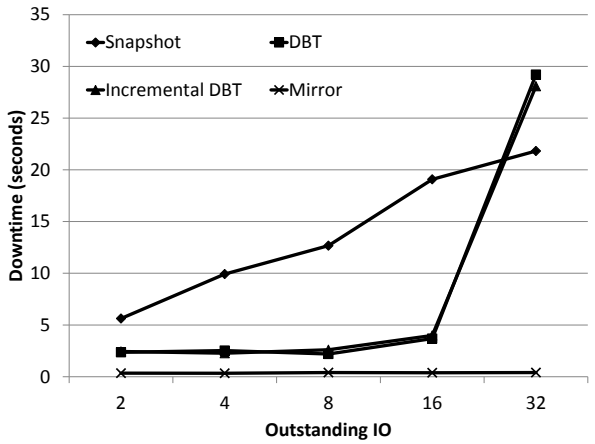


Figure 6: Downtime vs. OIO. IO Mirroring exhibits near constant downtime below 0.5s. DBT variants exhibit moderate downtime until OIO is 32 when the DM becomes a bottleneck. Snapshotting exhibits the worst downtimes except when OIO is 32.

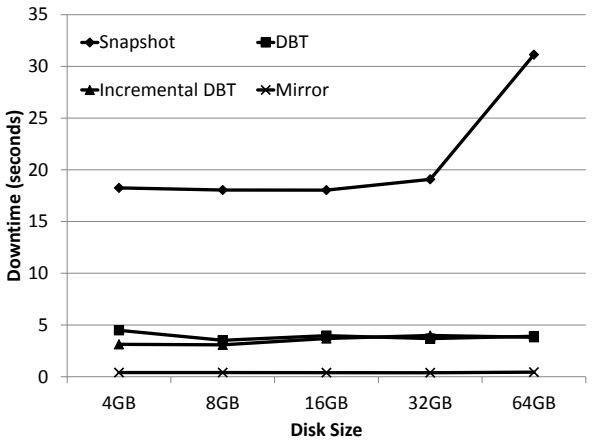


Figure 7: Downtime vs. Disk Size. IO Mirroring exhibits near constant downtime under 0.5s. DBT variants exhibit downtimes below the 5s convergence logic threshold. Snapshotting shows significant degradation beyond 32GB as snapshot overheads become prominent.

in migration time compared to DBT. IO Mirroring reduces the migration time by 52.7% compared to DBT. IO Mirroring is the only architecture to complete the double intensity workload successfully. The migration time appears lower, but the 5% difference is within the noise margin.

Overall, IO Mirroring exhibits the least change across workload intensity and disk size variations, while DBT and Snapshotting migration times increase significantly with such variations.

3.2 Downtime

While outages up to five seconds and beyond can be tolerated by some applications, others such as highly available applications, audio and video make even sub-second outages noticeable. Therefore we prefer to minimize downtime.

Downtime vs. OIOs is shown in Figure 6. With IO Mirroring downtime increases with increasing OIO by one tenth of a millisecond. This slight increase is due to the additional time required to quiesce the VM IO. There is no other downtime dependence on OIO for this architecture.

Both DBT variants choose their final copy thresholds with the intention of keeping downtimes under five seconds. Usually the algorithms overestimate, putting downtime consistently in the two to three second range for OIO under 8. From 8 to 16 we see that the downtime increases slightly as the DM begins to struggle to keep up. From 16 to 32, we see that the downtime jumps to values greater than 28 seconds.

The reasons for this are two fold, first the DM becomes a serious bottleneck however, our convergence logic also takes into account total migration time. If we were willing to wait an additional, potentially much longer time to converge, we might end up with a smaller final dirty page set, resulting in a shorter downtime.

Snapshotting shows a near linear growth in downtime as the workload increases, better than the DBT variants for the OIO equal to 32 case because the snapshot approach is consolidating the final snapshots on the destination volume.

Figure 7 shows the downtime as a function of disk size. Both DBT and IO Mirroring scale well with disk size. Snapshotting shows significant growth in downtime and total migration time when moving 64GB disks. As disk size increases, each snapshot create and consolidate iteration takes longer since there is increased disk fragmentation, increased virtual disk block lookup overhead, and other overheads related to the implementation of snapshots that accrue. Migration times also increase because the number of snapshot create and consolidate operations has to increase to keep downtime low.

Our Exchange workload shown in Table 1 exhibits larger downtimes for DBT and incremental DBT of roughly 8 and 13 seconds. For the double intensity workload, the DBT variants do not converge. IO Mirroring completes the migration for both the normal and double intensity workloads with roughly 0.1s and 0.2s of downtime.

IO Mirroring guarantees small constant downtimes. DBT variants offer low downtimes if the DM can keep up with the workload, but a slow destination volume or high

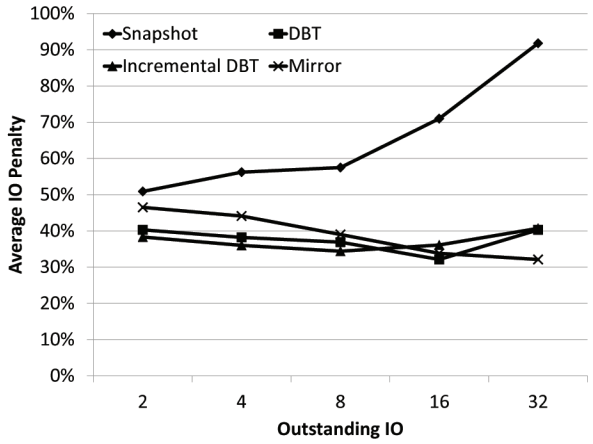


Figure 8: Average Guest IO Penalty vs. OIO. IO Mirroring exhibits lower penalties as OIOs increase as guest IOs come to dominate DM IOs. DBT variants exhibit similar pattern until 32 OIO when iterative copy times become an increasing portion of the overall cost. Snapshotting costs increase consistently as snapshot overheads worsen with increased OIO.

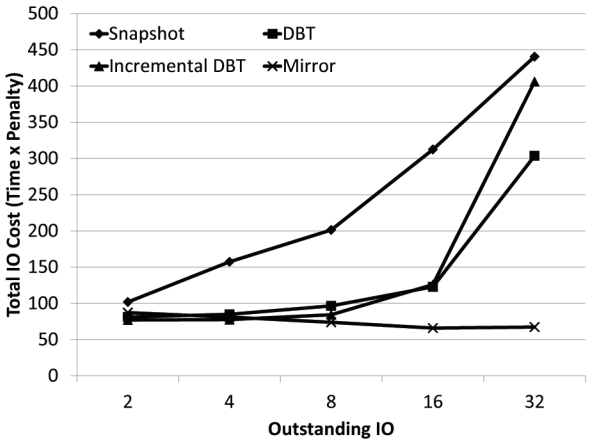


Figure 9: Aggregate migration cost (*total migration time* × *instantaneous penalty*) vs. OIO. IO Mirroring exhibits the lowest aggregate migration costs as OIO increases. As migration times stay near constant, average penalty decreases. DBT variants exhibit higher costs especially for 32 OIOs since migration times increase significantly. Snapshotting exhibits the greatest penalty as migration time and instantaneous penalty grow with OIO.

intensity workload may make that impossible. Snapshotting tends to have the highest downtimes.

3.3 Guest Performance Penalty

Minimizing the guest IO penalty and total cost during a migration improves user experience and lessens the impact on application IO performance. Average guest IO penalty vs. OIO for our synthetic workload is shown in Figure 8. The percentage IO penalty is identical for read and write IOs.

Snapshotting incurs the largest penalty because it uses snapshots, where IOs issued to new blocks require allocation and maintenance of indirection data structures. As OIOs increase, these overheads increase proportionally. If the workload runs long enough, these penalties will eventually start to amortize. However, since migration times are relatively short, these penalties remain high.

Penalties for DBT variants hover around 39%, due to guest IO competing with IO induced by the DM. With increased OIO, guest IO takes an increasing share of IO away from the DM. This causes slightly longer migrations. We observe slightly less IO penalty on the guest up until 16, since the DM has a maximum of 16 OIOs.

At 32 OIO the DBT variants show an increased IO penalty. The average instantaneous penalty is a combination of the penalty during the initial copy and the subsequent iterative copy phases. In the iterative copy phase, the IO penalty tends to be much higher than in the initial copy phase, because the DM is inducing a random IO workload on both volumes. In the 32 OIO case, the migration time increases significantly as shown in Figure 4. Consequently, the iterative copy phase contributes substantially more to the instantaneous penalty.

IO Mirroring begins with a penalty greater than both DBT variants but less than Snapshotting. Figure 8 shows that, as the workload OIOs increase the IO penalty becomes smaller than that of the DBT. The IO Mirroring architecture does not contain an iterative copy phase, thus it does not suffer from the higher performance penalties that DBT does with 32 OIOs. The only impact on the workload, assuming the destination is not a bottleneck, is due to the DM, we have a monotonically decreasing plot for performance penalty, because the workload consumes a proportionally larger share of the throughput.

Figure 9, shows the total guest penalty vs. OIOs. Penalty is measured in units of time, the total IOPS lost as if the workload was not executing for that many seconds. This shows that Snapshotting incurs the worst penalty amongst other migration implementations. The DBT variants both have similar migration penalties. Finally, IO Mirroring starts off with a penalty very similar to that of the DBT variants and gradually decreases as OIOs of the workload increase.

Exchange workload runs initially did not appear to show any degradation in throughput or IOPS. A closer look revealed that while average read latency remained the same, there was a slight increase in average write latency values per IO. However, write latency did not im-

Type	Latency	Variance	Penalty
No migration	0.777 ms	0.106	-
DBT	3.622 ms	0.363	4.7x
Incremental DBT	3.571 ms	0.544	3.6x
Mirror	3.338 ms	0.362	3.3x
Iterative Copy Phases			
DBT	5.922 ms	1.550	6.6x
Incremental DBT	5.171 ms	1.468	5.7x

Table 2: Comparison of changes in write latency observed on the Exchange 2010 workload.

pact IOPS or throughput, because the array could still consume the increased IO rate during the migration operation.

In Table 2, we list the write latency values observed during the migration of Exchange 2010 workload. The first number corresponds to the no migration case. The next three numbers corresponding to the migration architectures are obtained from the same amount of samples, during the copy phase of the disk. IO Mirroring completes as soon as the copy phase completes. The last two latency values are observed during the iterative copy phase of the DBT and incremental DBT methods, where the incremental DBT completes sooner than the DBT as shown in Table 1.

3.4 Convergence

We define convergence as an architecture's ability to reduce the number of blocks that differ between a source and destination to a level where a migration can complete with an acceptable downtime. We discuss convergence in our previous experiments, then examine a migration from a faster to slower volume, which is a challenging case for DBT variants. We omit Snapshotting from our discussion, as it always converges, but may still incur significant downtimes.

Every migration of the synthetic workload in the previous sections completes successfully however, when run with 32 OIOs, both DBT variants shows excessive downtimes and migration times. Downtimes larger than five seconds imply that the DBT convergence detection algorithm notices when the workload is not converging fast enough or at all, and considers aborting the migration. When the expected downtime is short enough the convergence logic forces the migration to complete, inflicting that downtime on the guest.

We know that DBT variants are not guaranteed to converge. For DBT to converge, it requires that the workload dirties blocks at a slower rate than the copy rate, and the destination volume should not be slower than the source.

An example of our DBT variants being overwhelmed

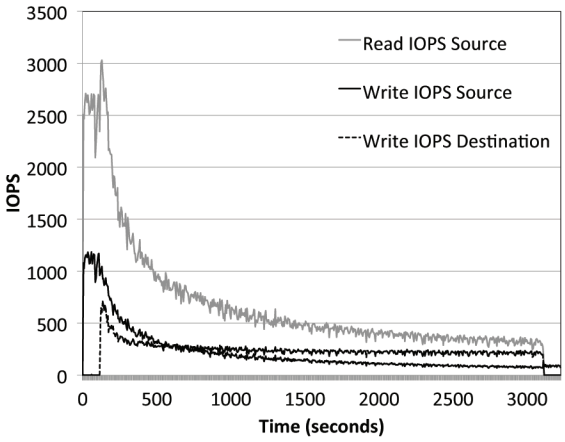


Figure 10: Graceful throttling of the 16 OIO synthetic OLTP workload during a migration from an FC to a slower NFS volume. The workload begins with approximately 2600 read and 1100 write IOPS and at the destination the workload runs at 200 write and 95 read IOPS.

is found among the 2x intensity Exchange workloads in Table 1, where neither variant completes successfully. Nevertheless, IO Mirroring completes the migration with negligible downtime. When running the normal intensity workload, all architectures complete their migrations successfully, but with significant downtime for the DBT variants.

In contrast, IO Mirroring converges even with a 10x slower destination volume. This is shown in Figure 10, where the total source and destination read and write IOPS observed during a migration from a Fibre Channel attached volume to a slower NFS attached volume, using our synthetic OLTP workload with 16 OIOs. The graph also shows that when IO Mirroring is involved, the gradual slow down is not linear but instead governed by the saturation of the OIO on the destination.

The left hand side of the graph shows the starting IOPS of the workload, which is approximately 2500 for read and 1000 for write. The shape of the gradual diminishing of IOPS may not be immediately clear. One may think that as an increasing percentage of IOs are mirrored the workload should also slow down linearly. That is not the case, because when the latency of the destination is more than an order of magnitude slower, the IOs on the destination will saturate the device. The mirror driver waits for write IOs on the destination to complete, while IOs on the source get acknowledged much faster. If we were to examine the number of OIOs at each device we would see that the source has only a few IOs while the destination is full. Equation 1, allows us to compute the approximate IOPS as a function of the OIOs, latency, and percentage of IOs sent to the destination (*DestIOPct*).

$$IOPS \approx Min(\frac{OIO}{DestIOPct \times Lat_{Dst}}, \frac{OIO}{Lat_{Src}}) \quad (1)$$

The migration starts after the first two minutes of stable IO in the graph, with a dip due to the IO Mirroring instantiation, followed by a rapid increase in read IOPS on the source. The read IOPS consists of the aggregate VM and migration read IOs. Similarly, the destination IOPS consists of storage migration and VM mirror write IOs. At the end of the migration, the VM IO workload stabilizes at approximately 200 read and 95 write IOPS.

When we run this benchmark using DBT, the migration fails after several iterations because it is unable to converge as the workload dirties blocks at a rate faster than the copy rate. Using IO Mirroring the workload is naturally throttled to the performance of the destination volume.

3.5 Anatomy of a Migration

Looking at IOPS over time provides unique insights about migration behavior. Figure 11, shows Time vs. IOPS seen by the host for all three migration architectures. The labels *S* and *D* refer to the region where the VM is the sole IO source. The initial disk copy is marked by the region *M*. In all three graphs the throughput for the initial disk copy is roughly the same. This suggests that the time it takes to clone a virtual disk has a constant cost, where enhancements such as hardware off-loading may be helpful.

With Snapshotting, while the disk is being copied, an initial snapshot is created on the source volume. Label C_S corresponds to the process of consolidating this initial source snapshot into the base disk. In region C_S the VM is running on a destination snapshot. In region C_D, we are creating and consolidating multiple snapshots on the destination. Each spike in IOPS marks the consolidation of the previous snapshot, and we see the slight degradation in random read/write IO as we access deeper offsets of a snapshot, until the next snapshot is created. Overall IOPS also increases due to the use of smaller snapshot sizes, resulting in few disk seeks.

DBT has a slightly better utilization of IO during the disk copy, because only a dirty block tracking bitmap is maintained in the kernel. In the region marked *DBT*, the iterative copy process begins. We see a reduction in throughput because the iterative copy process consists of random access to dirtied blocks. Together with the random IO workload, the total disk access pattern on the source becomes more random. Incremental DBT, which is not shown in Figure 11, shrinks the region *DBT* considerably with the optimization previously explained in Section 2.3.

IO Mirroring consists of a single pass copy, represented by region *M*. In this region, IO Mirroring shows a slightly lower utilization towards the end of the migration, because within the copy process, write IO is also mirrored to the destination volume. By the end of the migration all IOs will be mirrored. This method is shorter since there is no copy process that follows.

4 Related Work

Clark et al. implemented VM live migration on top of the Xen virtual machine monitor (VMM) [7]. Xen uses a pre-copy approach to iteratively copy memory pages from source host to destination host. It supports both managed and self migrations. Managed migration is performed by migration daemons running in the management VMs on the source and destination hosts, while self migration places the majority of the implementation within the guest OS of the VM being migrated. The guest OSes running on Xen are para-virtualized and are aware that a migration is in progress. The guest OS is able to improve migration performance by stunning rogue processes and freeing page cache pages. The authors mention, as possible future work, leveraging remote mirroring or replication to enable long distance migrations.

VMware VMotion [2] was first introduced in VMware ESX hypervisor and VirtualCenter suite in 2003. It supports the live migration of VMs among physical hosts that have access to shared storage, such as *storage area network* (SAN) or *network attached storage* (NAS). In VMotion, only VM memory pages are transferred from source host to destination. The virtual disks of a VM have to reside on shared storage and cannot move. VMware VMotion is the basic building block for VMware's Distributed Resource Scheduler (DRS) that dynamically balances the workloads between a set of hosts in a cluster, and for Distributed Power Management, that uses migration to consolidate workloads to reduces power consumption during off-peak hours [1].

In addition to the pre-copy approaches, Hines et al. proposed a post-copy approach for live VM migration [8] for write-intensive workload by trading off fast migration time with atomic switchover between source and destination hosts. Pre-paging and self-ballooning mechanisms are also used to facilitate the post-copy approach. Pre-paging utilizes locality of access to reduce the number of network page faults. Self-ballooning is technique to remove unneeded buffer cache pages from the guest so they will not be transfered.

VM live migration has also been extended from *local-area networks* (LAN) to *wide-area networks* (WAN) for various use cases, including data center load balance and disaster recovery. Bradford et al. extends the live migration in Xen to support the migration of a VM with

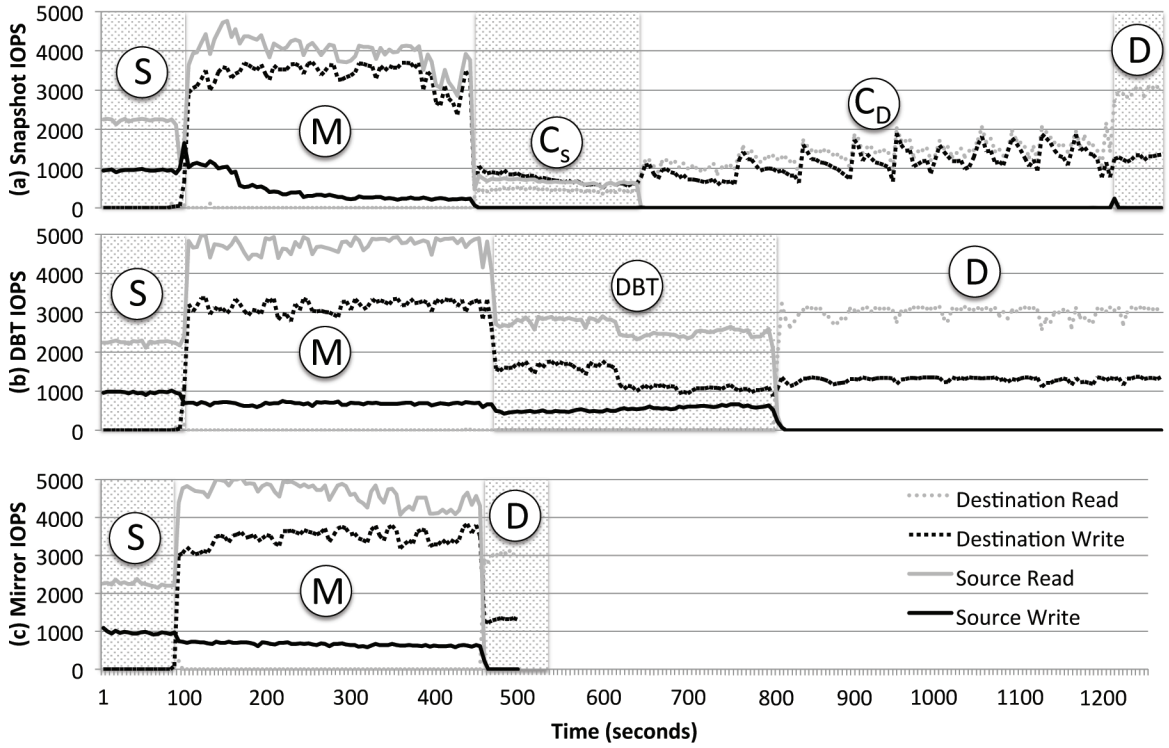


Figure 11: The IOPS observed for the duration of the three main architectures. region *S*, is prior to the migration, and region *D*, corresponds to the post migration. Region *M* in all graphs refers to the copying process of the virtual disks. In the Snapshot graph, the area denoted as C_S and C_D correspond to the consolidation of the source and destination snapshots. Finally, the shaded region *DBT* refers to the iterative copy operation.

local disk states across WAN [9]. When a VM is being migrated, its local disks are transferred to destination volume using a block level disk pre-copy. The write IO workload from the guest OS is also throttled to reduce the dirty block rate. Existing network connections to the source VM are transparently forwarded to the destination by using IP tunneling, while new connections are redirected to the destination network using Dynamic DNS. Further optimizations for wide area storage migration are explored by Zheng et al. [10]

VM Turntable demonstrates live VM migration over WAN links for Grid computing [11]. In this system, dedicated paths are dynamically setup between distant sites. However, no local disk state is transferred over the WAN in VM Turntable. CloudNet builds a private network across different data centers using MPLS based VPN for live WAN migration of VMs [12]. Disk state for VMs is replicated to the destination volume using both asynchronous and synchronous schemes in different migration stages.

Storage mirroring or disk shadowing is an existing concept that was first explored for building redundant disks [13]. Mirroring for redundancy and performance reasons is very common in volume managers such as LVM [14] and Vinum [15]. Network based mirroring for replication and virtually shared volumes that span multiple physical hosts is done by distributed block devices like DRBD [16] and HAST [17].

Live storage migration of VMs using IO Mirroring is discussed in Meyer et al. [18]. This work presents a modular storage system that allows different storage systems to be implemented by configuring a set of components.

SAN vendors provide volume level mirroring and replication for redundancy such as NetApp's SnapMirror product [19] and EMC's Synchronous and Asynchronous SRDF [20]. The concept of LUN migration is also provided by some of the storage array vendors that is similar to the problem of virtual disk migration. For example, Data Motion [21] from NetApp uses asynchronous SnapMirror to initiate the copy and achieve a small target *recovery point objective* (RPO) time. During the final switchover the NetApp LUN may be unavailable for up to 120 seconds. If the Data Motion is unable to complete within that time it cancels the LUN migration [21]. The NetApp Data Motion product functions similar to dirty block copy approach discussed in Section 2.3 and does not appear to meet our goals of no downtime and guaranteed success.

5 Conclusions

We present our experience with the design and implementation of three different approaches to live storage migration: Snapshotting (in ESX 3.5), Dirty block tracking (in ESX 4.0/4.1) and IO Mirroring (in ESX 5.0). Each design highlights different trade-offs by way of impact on guest performance, overall migration time and atomicity.

The first two approaches exhibit several shortcomings that motivated our current design. Snapshotting imposes substantial overheads and lacks atomicity, this hinders reliability and makes long distance migrations fragile. DBT adds atomicity, and by working at the block level allows a number of new optimizations, but also cannot guarantee convergence and zero downtime for every migration.

While the IOPS penalty caused by Snapshotting to the OLTP workload is around 70%, DBT and IO Mirroring reduce this penalty to around 32% and 34%. The total penalty for IO Mirroring is approximately 2x better than DBT.

Our latest approach based on IO Mirroring offers guaranteed convergence, atomicity and zero downtime with only a slightly higher IOPS penalty than DBT. When moving a virtual disk to a slower volume, IO Mirroring exhibited a graceful transition and completion. We achieved consistent reduction in total migration time, bringing the total live migration duration close to that of a plain disk copy. For the OLTP workload, IO Mirroring, takes less than half the time of Snapshotting and only 9.7% longer than an off-line virtual disk copy. We showed that under varying OIO and disk size, IO Mirroring offered very low variation in migration time, downtime, and guest performance penalty.

Acknowledgements

We would like to thank Vincent Lin for running Exchange workloads; Irfan Ahmad, Joel Baxter, Dilpreet Bindra, Kit Colbert, Christian Czezatke, Ajay Gulati, Jairam Ranganathan, Mayank Rawat, Yan Tang, Jayant Kulkarni, Murali Vilayannur, Swathi Koundinya and Chethan Kumar for their valuable discussions and feedback with the original work.

References

[1] "VMware Infrastructure: Resource Management with VMware DRS," Aug. 2010. `http://www.vmware.com/pdf/vmware_drs_wp.pdf`.

[2] M. Nelson, B.-H. Lim, and G. Hutchins, "Fast transparent migration for virtual machines," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, (Berkeley, CA, USA), pp. 25–25, USENIX Association, 2005.

[3] "vstorage api for array integration and vmware ready," `http://www.vmware.com/partners/programs/vmware-ready/vstorage-api-arrays.html`.

[4] "Exchange Load Generator." `http://technet.microsoft.com/en-us/library/bb508866(EXCHG.80).aspx`.

[5] Cristian Estan, et. al., "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice," *ACM Transactions on Computer Systems*, 2003.

[6] "Iometer." http://www.iometer.org.

[7] C. Clark, et. al., "Live Migration of Virtual Machines," in *NSDI*, Oct 2005.

[8] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 14–26, 2009.

[9] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, "Live wide-area migration of virtual machines including local persistent state," in *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, (New York, NY, USA), pp. 169–179, ACM, 2007.

[10] J. Zheng, T. S. E. Ng, and K. Sripanidkulchai, "Workload-aware live storage migration for clouds," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '11, (New York, NY, USA), pp. 133–144, ACM, 2011.

[11] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Y. Wang, "Seamless live migration of virtual machines over the man/wan," *Future Gener. Comput. Syst.*, vol. 22, no. 8, pp. 901–907, 2006.

[12] T. Wood, K. Ramakrishnan, J. van der Merwe, and P. Shenoy, "Cloudnet: A platform for optimized wan migration of virtual machines," *University of Massachusetts Technical Report TR-2010-002*, January 2010.

[13] D. S. Dina, D. Bitton, and J. Gray, "Tandem tr 88.5," 1988.

[14] "Logical volume manager (linux)." `http://en.wikipedia.org/wiki/Logical_Volume_Manager_(Linux)`.

[15] "The vinum volume manager." `http://www.vinumvm.org/`.

[16] "Distributed Replicated Block Device (DRBD)." `http://www.drbd.org/`.

[17] "Hast - highly available storage." `http://wiki.freebsd.org/HAST`.

[18] D. Meyer, B. Cully, J. Wires, N. Hutchinson, and A. Warfield, "Block mason," in *WIOV '08: First Workshop on I/O Virtualization*, 2008.

[19] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara, "Snapmirror: File-system-based asynchronous mirroring for disaster recovery," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, (Berkeley, CA, USA), p. 9, USENIX Association, 2002.

[20] "EMC SRDF Datasheet." `http://www.emc.com/collateral/software/data-sheet/1523-emc-srdf.pdf`.

[21] L. Touchette, R. Weeks, and P. Goswami, "Netapp data motion," *NetApp Technical Report TR-3814*, March 2010. `http://media.netapp.com/documents/tr-3814.pdf`.

Online migration for geo-distributed storage systems

Nguyen Tran* Marcos K. Aguilera Mahesh Balakrishnan
Microsoft Research Silicon Valley

Abstract

We consider the problem of migrating user data between data centers. We introduce *distributed storage overlays*, a simple abstraction that represents data as stacked layers in different places. Overlays can be readily used to cache data objects, migrate these caches, and migrate the home of data objects. We implement overlays as part of a key-value object store called Nomad, designed to span many data centers. Using Nomad, we compare overlays against common migration approaches and show that overlays are more flexible and impose less overhead. To drive migration decisions, we propose policies for predicting the location of future accesses, focusing on a web mail application. We evaluate the migration policies using real traces of user activity from Hotmail.

1 Introduction

Internet web applications are increasingly important to our everyday lives, as we rely on them for email, searching, online storage, online calling, and much more. These applications face a data scalability challenge that is getting worse, for two reasons. First, there is a growing number of users in an increasing number of regions. And second, the storage needs *per user* are growing as more applications become available online, users accumulate more data, and systems collect more information from users to target ads and personalize their experience. As a result, these applications need to be *geo-distributed*, which means they are deployed across multiple data centers around the world, due to constraints on the size, bandwidth, and power consumption of a single data center. Besides providing scalability, geo-distribution also allows a user to be served from a nearby data center, thereby reducing user response times and bandwidth consumption. For that, the user's data should be at the right data center, namely, a data center close to the user. This is called *access locality*.

Unfortunately, data is not always where it should be: users relocate, and the load at data centers becomes unbalanced due to new applications, new data centers, and changes in the network topologies. In these cases, user data needs to be migrated from one location to another; migration is essential to provide access locality and to balance load. This paper considers the problem of migrating data across data centers. We propose a simple abstraction called *distributed data overlay* or *overlay* in short[1], which represents data as stacked layers stored in different places. Overlays are a flexible way to support data migration; they can be used to cache data at remote data centers, migrate these caches from one data center to another, and migrate the *home* of a data object—the data center where the object is stored when it is not cached. If data is replicated across data centers, overlays can be used to migrate individual replicas.

With overlays, migration can be performed *online*, that is, while the data is accessible to users. This is important for three reasons. First, user data can be massive and the bandwidth across data centers is limited, so that migration can take a long time and we do not wish to disable the user account during migration. Second, we want to migrate data opportunistically in the background, using possibly small amounts of left-over bandwidth. This is so because large companies such as Microsoft pay for private links with fixed bandwidth to connect data centers, which means that unused bandwidth is wasted money. Third, the policies of when to migrate data can be complex, and we do not want to complicate them further with constraints and predictions of when users will access their data.

Online migration is challenging due to races; it requires careful coordination as clients in the network read and write data while the migration process copies the data and the system possibly creates, flushes, and removes caches at remote locations. Overlays are an easy, flexible, and efficient way to handle this coordination, as we demonstrate in this paper.

We implement overlays in a system called Nomad, which is a key-value object store that supports online migration. Key-value stores were recently proposed to support large-scale applications in data centers (e.g., [16]). Though Nomad is a key-value store, overlays are applicable to other types of storage, such as distributed linear-address stores [9], block stores [31], and file systems.

We evaluate the mechanism for migration using a wide-area deployment on five data centers around the world. Our experiments show that overlays impose a small overhead and provide flexibility for supporting caching and migration. They also show that overlay-based migration is more efficient than existing methods based on data locking and logging.

The mechanism for migration is independent of the policies used to trigger migration. We study some simple policies that track the location of users as they move. We evaluate these policies using real traces of user accesses; we compare policies based on access count, time,

*Current affiliation: New York University

[1] not to be confused with a network overlay.

and rate, and we show that, although they are all reasonable, the one based on count performs the best.

Summary of contributions. We consider the problem of building distributed storage systems deployed over many data centers, with support for flexible online migration of data across data centers. We propose distributed data overlays, a simple but flexible abstraction designed to hide the complex distributed protocols (which we provide) required to coordinate access to data at many locations. We also propose policies for driving the migration of user data, and evaluate them using real traces from Hotmail. We implement overlays to produce the Nomad system, and use it to compare our approach against less flexible but common alternatives for storage migration.

2 Background

There are many data centers around the world, each with thousands or more machines, subject to crash failures. We do not consider Byzantine failures in this paper. We target a setting where partitions across data centers are rare in the absence of disasters. This can be achieved by connecting data centers via private leased lines with high availability [1, 2]; by using redundant links to maintain operation during planned link downtime (e.g., using a ring topology across data centers); and by routing traffic via the Internet should all the redundant links become unavailable. The data centers run web applications that store user data, such as these:

Application	User data
web mail	emails
web phone	voice mails
web storage	personal files
chat	text/image history
search	search history
ALL	profile, activity logs

This data should often be stored at a data center closest to the user, where the user logs into. Migration refers to moving the data from one data center to another—to improve access locality or to balance load across data centers. *Online* migration means that, during migration, the data remains accessible to the applications.

We now illustrate some migration use cases with three scenarios; we later explain overlays and how they support these use cases. In these scenarios, "user data" refers to the data specific to a user that an application needs to serve that user. For example, it could be the user's emails.

Scenario 1 (A long trip to China): A French user goes to China. After several days, the system starts to migrate her user data to China. If she goes back, the French copy is updated with any changes made in China. If she stays in China longer, all her data is migrated and the French copy is deleted.

Scenario 2 (Backpacking in Asia): The French user makes a short trip to China and, soon after, the system creates a cache at a data center in China containing her recent user data (e.g., recent emails). She then travels to Russia, and so the system migrates the cache in China to a data center in Russia. She stays in Russia for some time, and so the system starts to migrate her data from France to Russia, which takes several days. Before the migration is over, she returns to France, so the system applies all updates done in China and Russia to her data in France.

Scenario 3 (Data center expansion): A data center in France is nearing maximum storage capacity, and so a data center in Spain is created and the system migrates some users from France to Spain. During this migration, the two above scenarios may happen with some of the users being migrated from France to Spain.

More generally, migrations can be *ephemeral* or *permanent*. Ephemeral migrations are reversed in the future; they are implemented by creating a cache of the user data at a new location and possibly pre-fetching parts of the data. Later, the cache is flushed if it has dirty data and then removed. Permanent migrations are not reversed; they are implemented by copying the user data to the new location, while coordinating updates to the data so that they go to the right location. Sometimes, a migration may start off as being ephemeral, but may end up being permanent—this could happen, for example, if a user travels to location and ends up staying there for the rest of her life. In that case, the cache gets transformed into the home of the data. Ephemeral and permanent migrations may occur simultaneously, say because the user is traveling but her home data center is being reassigned.

Migration must functionally appear as a no-op: reads and writes should functionally behave the same way whether or not migration has occurred or is in progress, except in terms of performance (a completed migration will improve performance by reducing the number of remote accesses). Moreover, migration is an optimization rather than a task required for correct operation of the system. We do not wish migration to disrupt the performance of the system by consuming large amounts of bandwidth during busy times. We thus expect migrations to occur in the background with low priority.

3 Distributed data overlays

In this section, we describe our approach to migration using distributed data overlays or simply *overlays*. Our description is targeted at a fairly general distributed storage system. In Section 4, we provide the details of overlays for a specific key-value store system called Nomad.

Overlays are an abstraction to provide online migration. Migration results in partial copies of data at two or more locations—such as cached fragments and partly-copied data—which need to be managed carefully while the system orchestrates accesses, to ensure writes are not lost and reads return valid data. For example, if data is written at the old location at the same time it is being

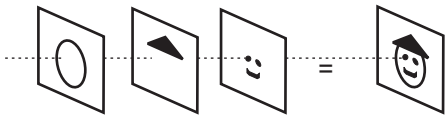


Figure 1: Overlays.

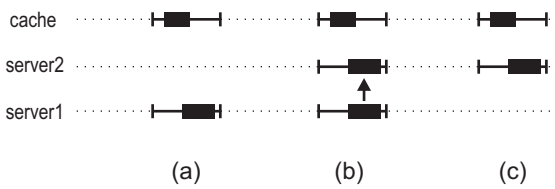


Figure 2: It is easy to use overlays to migrate data from server1 to server2. The black bars indicate regions with data. (a) Initially, data is in server1, and there is a dirty cache at a data center close to the user. (b) First, insert an overlay at server2 between server1 and the cache, and copy data from server1 to server2. (c) Then, remove the overlay at server1; dirty cache remains in place. The distributed protocols that implement overlays ensure that the insertion and removal of overlays will never cause the loss of data in ongoing read or write operations.

migrated to a new location, the system may fail to migrate the new write. This example becomes more complex when there are dirty caches, those caches themselves are being migrated, and/or migrations are canceled and restarted; the number and complexity of the different scenarios that must be handled can be problematic for the system developer. Overlays are an abstraction that helps dealing with these scenarios in a simple and unified way.

As an everyday analogy, an overlay is a sheet of transparent plastic that is placed over a piece of paper. Where it is clear, the overlay reveals the contents underneath; where it is written, the overlay overrides those contents (Figure 1). Overlays can be stacked, to create many layers, so that looking at the stack reveals their combined contents; if many overlays have content at the same place, the higher overlays occlude the lower ones.

This idea has an analogue to storage systems. We now explain it in a context where the user data is a byte sequence, such as a file, a data object, or the sequence of blocks on a disk—depending on the nature of the storage system. Data is stored at some base location and it may be partly stored in another data center, which serves as a cache. We can view the base location and cache as a stack of two overlays, as shown in Figure 2(a), where each overlay is stored in a server in a data center. For uniformity, the base layer is also called an overlay. The combination of all overlays determines what data is seen on the stack, with higher overlays having priority over lower overlays.

With the abstraction of overlays, migrating data is straightforward: (1) we create an overlay below the caching overlay, residing at the destination server, (2) we populate the new overlay by copying data from the base overlay, (3) we delete the base overlay, so that the new overlay becomes the new base (see Figures 2(b) and 2(c)).

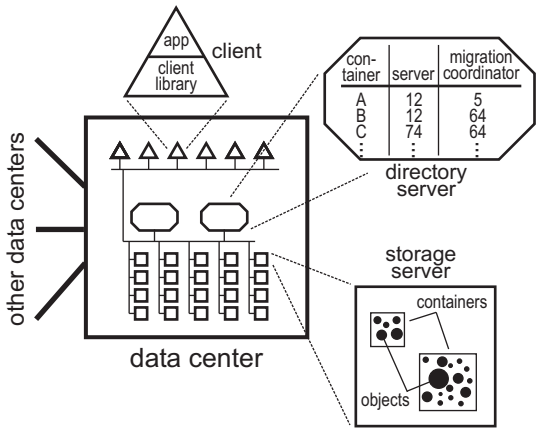


Figure 3: Standard object store architecture used in Nomad.

Function	Description
read(*container*, *oid*, *off*, *len*, *buf*)	read object
write(*container*, *oid*, *off*, *len*, *buf*)	write object
create(*container*, *oid*, *len*, *buf*)	create and write object
delete(*container*, *oid*)	delete object

Figure 4: Nomad API to access objects.

While migration occurs, data can be written at the cache, and the cache can be flushed by writing its contents to the new overlay. The protocols implementing overlays (which are hidden from the designers who just want to use them) ensure that reads and writes on an overlay stack go to the right overlay, and that overlays can be created, inserted, and removed atomically even if reads and writes occur concurrently. With overlays, it is easy to support the three scenarios described in Section 2, by inserting overlays for caches or other copies of data, and copying data between overlays to migrate. Note that each overlay is kept at a fixed server, that is, an overlay does not move; migration is achieved by creating overlays and copying data between them.

4 Nomad design

We built Nomad, a prototype of a distributed key-value object store that incorporates overlays to support flexible and online migration in a geo-distributed setting. We describe overlays in Nomad for concreteness; however, overlays are applicable to other types of storage systems, such as file systems or block storage systems.

4.1 Basic architecture

Nomad has a typical architecture for a distributed key-value object store, shown in Figure 3. This architecture is not novel; we describe it in this section for completeness.

Objects are stored on a set of storage servers, which are commodity machines running a standard operating system; they store each object as a separate file in the local file system. Throughout the paper, *client* refers to the entity that uses Nomad, which is an application run-

ning at a server in the data center, whereas *user* refers to the entity that uses the application, which is often outside the data center—for example, the user could be a person using a web mail system. Clients access Nomad via a client library that implements functions for reading, writing, creating, and deleting objects, as shown in Figure 4. There are also functions to read or write multiple objects in the same request for efficiency; these are not shown for simplicity. This interface is simple: a write associates a key with a new value, and a read returns the latest value associated with the key. Each object is part of a *container*, similar to a directory in a file system, a bucket in Amazon's S3 [5], or a blob container in Microsoft Azure [6]. A container is stored in one of the storage servers, and there is a function for enumerating the object identifiers in a container, and to create/remove containers (not shown). The mapping of containers to storage nodes is kept by a directory service, which is replicated asynchronously across data centers. For flexibility, Nomad allows any container to be mapped to any storage server; the mapping is represented as a list of container-server pairs. The client library caches part of the mapping, and the directory service need not keep track of who caches what: if the mapping changes (because the container is migrated), the client library may try to access an object at the wrong location, in which case the client library gets an error, consults the directory to find the right location, and tries again. It is possible to expire entries in the client cache for efficiency, so that the client does not use too old information, but Nomad does not do that. The directory service also indicates a *migration coordinator* for each container (Section 4.5).

4.2 Migration granularity

At what data granularity should migration occur? We discuss this issue from three perspectives: the application, the system administrator, and the storage system.

From an application perspective, applications should organize and migrate data in units that are likely to be accessed together within a given location, to provide locality. For example, in a web application, a user usually logs in at a data center close to where she lives; her persistent data, such as personal information and preferences, form a coherent unit for migration. It would not make sense to migrate a user's name without migrating her address, for example.

From the system administrator's perspective, the choice of granularity comes from a balance of manageability and control. On one hand, migration should be fine enough to allow reasonable control over the allocation of storage capacity and bandwidth. On the other hand, migration should be coarse enough so that the number of units to be administered is small.

From the storage systems perspective, the migration granularity should match the granularity of the mapping at the directory service, so that the migration engine does not have to reimplement this functionality. In particular, before migration, the directory maps some unit of data to some server; after migration, this unit must be mapped to a different server without leaving behind intermediate mappings.

Consequently, migrations in Nomad are done at the granularity of a container, and we intend that application designers collaborate with system administrators to choose an appropriate organization around such containers. For example, in some web applications, all of a user's personal information and preferences could be stored in a container. In a web mail system, there could be a container per email folder per user, so that containers are not extremely large.

A related consideration is the specificity of the destination of migration. In Nomad, the migration targets are servers, but a high-level migration decision by an administrator could be to move a container from one data center to another. In this case, there has to be a component that refines this decision and picks actual servers; this component, as well as the actual policies for migration, are orthogonal to the migration mechanisms in Nomad.

4.3 Overlays in Nomad

Our description of overlays in Section 3 assumed simplistically that the user's data and migration granularity is a sequence of bytes. We extend the description to Nomad, where the migration granularity is an object container, which consists of a set of objects, where each object is a sequence of bytes. An overlay for the container is an overlay for each object in the container plus an overlay for an array of bytes representing the set of object identifiers in the container. All objects in the container have identical overlays, except that the data contents for different objects are different. Thus, for efficiency, Nomad keeps a single *overlay structure* per container, which represents the (identical) overlays of all objects in the containers, without any data; the data is kept separately as a set of extents for each object at each overlay. An object may have several extents at a given overlay.

Overlay internal information. Recall that the directory service maps each container to a storage server, which in turn stores the base overlay for the objects in that container. Normally, the base overlay is the only overlay in the stack, but when the container is being migrated or cached, there may be additional overlays. The overlay structure consists of the following information:

- *container-id:* container that the overlay refers to;
- *location:* server that stores the data in the overlay;
- *above-pointer:* pointer to the overlay above, or nil;
- *below-pointer:* pointer to the lower overlay, or nil;
- *frozen:* a flag indicating that overlay pointers cannot be changed;

An overlay structure is associated with the following:

Function	Description
insert(*server*, *overlay*, *direction*, *flags*)	create overlay and insert
remove(*overlay*)	remove overlay
get_stack(*base_overlay*)	get entire overlay stack
start_copy(*overlay*, *direction*[, *list*])	copy to adjacent overlay
stop_copy(*copy_job*)	stop copying

Figure 5: Operations on overlays.

- *data:* a set of extents for each object, with a *unique bit* and a *timestamp* for each extent.

The unique bit is unset when the extent is a repetition of data in a lower overlay; this bit is similar to the dirty bit in a cache. The timestamp is used to handle concurrent writes when data is replicated at many overlays: the write with highest timestamp wins. We explain replication in Section 4.6.

Reading and writing data. To write data to an object, the client first finds the highest overlay O_{high}, by starting with the base location and successively traversing the *above-pointer* at each overlay until it becomes nil. Then, the client sends the data to be written to the overlay O_{high}. If the *above-pointer* at O_{high} remains nil, the storage server at O_{high} accepts the write and sets the unique bit for the newly written extent. (The checking that the above-pointer is nil and the acceptance of the write must be performed atomically with respect to the processing of other client requests for the overlay.) Otherwise, there has been a concurrent operation to insert an overlay above O_{high}, so the storage server returns an error together with the value of *above-pointer*; the client continues the traversal to find the new highest overlay, and retries the write there. When the client has completed the write, it caches the identity of the highest overlay it found. In its next write, the client starts the traversal from the cached overlay, for efficiency. The cached value could be an overlay that no longer exists (because it was removed), in which case the client gets an error and consults the directory service to find the base location again.

To read an object, the process is similar but slightly more complex, because the highest overlay may not hold the data to be read; in that case, the client goes back to the lower overlays until it finds the data it wants. It is possible that an overlay holds only part of the interval to be read, in which case the client goes to the lower overlays for the missing pieces.

Note that when there is a single overlay—which is often the case for most objects—its location is the server indicated by the directory service, and a read or write request proceeds as in a system without overlays, without additional communication rounds.

Overlay operations. The operations that insert, remove, and copy overlays are shown in Figure 5. The insert operation indicates the server for the new overlay, an existing overlay where the new overlay will be inserted, a direction (*above* or *below*) to specify whether to insert above or below the specified overlay, and a flag with properties for the new overlay. Currently, the only property is whether the new overlay holds unique data or not. If it does not, then when a write happens at the overlay, the write is also forwarded to the overlay below; this mechanism can be used to implement a write-through cache. The *start_copy* operation copies the objects in the overlay to the overlay above or below. It can copy all object or just those indicated on a list—this is useful to populate caches with certain objects only. The remove operation is self-descriptive; the system takes care of copying the overlay's unique data to the overlay below before removing it. It is not legal to remove an overlay if it is the only overlay in the stack. Not shown in the figure are the operations that return the base overlay for a container and for an object.

To simplify the design, we require that overlay operations be executed one at a time per container. This serialization occurs per container, not across containers, and so it does not pose a performance problem since overlay operations on a container are relatively rare. To serialize, overlay operations can be called by only one server per container: in Nomad, this server is indicated by the directory service and it is called the *coordinator* of the container. The coordinator ensures that an overlay has at most one outstanding overlay operation. To achieve fault tolerance, we can fail over the coordinator as we explain later. Note that read and write operations are *not* overlay operations: they can be executed concurrently with overlay operations and with each other, at many clients. The protocols that implement overlay operations, described in Section 4.5, ensure correct behavior in these cases.

4.4 Using overlays in Nomad

It is easy to use overlays to migrate data, create a cache, migrate the cache, and migrate data back, as we now describe. We provide intuitive explanations in English, but it is easy to translate these explanations into code that calls the functions in Figure 5.

Migrate data to another server. The system creates an overlay at the destination server on top of the source overlay to be migrated; at this point, writes will no longer go to the source overlay. The system then invokes the operation to copy the data from the source to the destination overlay. When the copy is finished, it removes the source overlay. As we mentioned, because we designed the overlay operations so that clients can concurrently access data, migration proceeds concurrently with these accesses, and without causing reads or writes to be lost.

Cancel migration. Sometimes, migration should be canceled because of changes in the workload. For instance, if a user is traveling for some time and migration starts, but the user returns before migration has finished, the system may decide to cancel the migration. This is

easy: we simply stop the copying operation and remove the new overlay that was created for migration. Recall that the operation to remove an overlay copies the overlay's content to the overlay below. If the new overlay already has lots of data, the following optimization is effective. Note that only data written by the client needs to be copied, not data written by the migration, which is already present in the overlay below. To identify these writes, the writes by the client have a special *unique* bit set (Section 4.3), while the writes by the migration do not.

Create cache at a data center. We can create two types of caches at a data center: write-back or write-through. (Caching can also be done at the client; this can be done by the application if desired, not by Nomad.) To establish a cache, one simply inserts a new top overlay stored in the desired data center; write-back or write-through behavior is indicated by the *flag* parameter of the insert operation, which indicates whether the overlay will forward writes to the overlay below or not. To flush the cache, one invokes the copy operation to the overlay below. To remove the cache, one invokes the operation to remove the overlay.

Migrate the cache. To migrate a cache (which may have dirty data), we use the procedure to migrate the data at an overlay, described above.

4.5 Implementing the overlay operations

We now describe how to implement the overlay operations of Figure 5. We make the following design decisions: (1) it is reasonable to serialize overlay operations on the same container, but we should allow operations on different containers to run in parallel, and (2) an overlay operation on a container must allow reads and writes on the container to proceed in parallel, because these operations are sensitive to performance. Therefore, we assign a *(migration) coordinator* per container, which executes overlay operations on that container one at a time, and we design the coordinator protocol carefully so that reads and writes are never blocked. The coordinator is indicated by the directory service, and it manipulates the overlay state at each server via remote procedure calls (RPCs), as we now explain.

Inserting overlays. To insert an overlay $O2$ at storage server S above overlay $O1$ and below overlay $O3$, $O1$ and $O3$ must point to $O2$, and $O2$ must point to both. To do so, the coordinator executes the following actions (using RPCs), in this order: (1) create $O2$ at S with pointers to $O1$ and $O3$; (2) change $O1$.above-pointer to $O2$; (3) change $O3$.below-pointer to $O2$. Note that after (2) before (3), $O2$ is already visible to read and write operations because $O1$ points to it, but $O2$ is in a funny state where $O3$ does not point to it yet. This is not a problem, because $O2$ has no data and it is impossible for it to get any data (writes would go to $O3$ instead).

To insert $O2$ at the top, the process is similar except that $O2$'s top pointer is nil, and step (3) above is skipped. To insert $O2$ at the bottom, the process is also similar except that $O2$'s bottom pointer is nil, and step (2) changes the base pointer at the directory service to point to $O2$. There one subtlety: the directory service is replicated asynchronously; the coordinator changes only the directory server in its own data center and the others are eventually updated; in the meantime, the remove directory servers may temporarily point to the wrong base; this is not a problem since the directory service is used only for finding the top overlay (see "Reading and writing data" in Section 4.3).

Removing overlays (part 1). To remove an overlay $O2$, we first consider the case when $O2$ is completely occluded by the overlay above: that means all data in $O2$ is covered by data at the overlay above, so that the data in $O2$ is useless. In that case, the coordinator can remove $O2$ without fear of losing data; to do so, the coordinator (1) changes the overlay below to point to overlay above, (2) changes the overlay above to point to the overlay below and sets the unique bit for all extents in the overlay above[2]. If there is no overlay below, because $O2$ is the base overlay, the coordinator changes the base pointer at the local directory server (instead of the overlay below); the other replicas of the directory server may temporarily point to the deleted $O2$, so we leave a tombstone at $O2$ pointing to the overlay above; the tombstone is removed after a period long enough that all directory servers have seen the update (say, one hour).

Another easy case is to remove $O2$ when the overlay below is in the same storage server. In that case, the coordinator asks the storage server to execute three actions: (1) locally copy the contents of $O2$ to the overlay below $O1$, (2) redirect any writes to $O2$ so that it goes to $O1$, and (3) make $O1$.above-pointer point to $O2$.above-pointer. These three actions can be done without races because they are done in the same server. Finally, if there is an overlay above $O2$, the coordinator makes its below-pointer point to $O1$.

The removal process we described so far does not allow one to remove the top overlay, or some overlay that is not completely occluded. We come back to that soon, because such an operation uses the next operation.

Copying data between overlays. To copy data from an overlay to the overlay *below*, the coordinator asks the server of the overlay above to send the data to the server below. This idea can also be used to copy from an overlay to the overlay *above*, but it is more efficient to ask for the overlay *above* to pull the data from the overlay below, because if the overlay above already has data for certain objects, these objects need not be copied (since the overlay above occludes the overlay below at those objects).

[2] Setting the unique bit this way is a conservative choice.

Removing overlays (part 2). We can now describe how to remove an overlay $O2$ that is not completely occluded. The procedure to do that reduces to invoking existing procedures that we already described. There are two cases:

1. If there is an overlay above $O2$, the coordinator invokes the copy operation from $O2$ to that overlay, to occlude $O2$. The coordinator then uses the previously-described procedure to remove an overlay that is occluded. This works without any races, because once an overlay is occluded, it remains occluded as no writes can go to it—unless the overlay above is removed, but as we explained above, this does not happen since all operations on an overlay are serialized by the coordinator.

2. If $O2$ is the top overlay then it must have some overlay $O1$ below it at some storage server S (the last overlay cannot be removed, which would result in data loss). The coordinator first creates a new temporary overlay $O3$ over $O2$ at storage server S. Then, the coordinator removes $O2$ using the above procedure, since $O2$ is no longer the top overlay. We are left with overlays $O1$ and $O3$ at server S. The coordinator now uses the above procedure to remove an overlay when the overlay below is in the same storage server.

Copy optimization. The coordinator serializes overlay operations, but this is inefficient in one case: the copy operation can take a long time and hence delay further overlay operations. For example, suppose that we have a base overlay and a cache, and we want to migrate the base from a server to another one. During this migration, we may want to also migrate the cache to another place, but if the overlay operations on the same container are serialized, the cache migration must wait for the server migration to finish, which is undesirable. We address this problem by having copy operations run in the background, thereby allowing concurrent execution of other overlay operations on the same container. For this to work, we need to restrict the other overlay operations so that they do not change the source and destination overlays involved in a copy (for example, it would be problematic to remove the source or destination overlay while the copying is going on). We do this simply by setting a "frozen" flag at the overlay; an overlay operation that encounters the frozen flag exits with an error and retries later. It suffices to freeze the lower of the two overlays, because the operations to remove the higher or lower overlay or to insert an overlay between them will access the lower overlay first and find the frozen flag.

Correctness proof. The operations to insert, remove, and copy data between overlays ensure that reads and writes behave equivalently as if they were executing in a *single-overlay system*, that is, a system that has a single fixed overlay where all the reads and writes occur.[3] As a consequence, read and write operations are linearizable [20], which provides a strong form of consistency. Roughly speaking, linearizability ensures that each operation appears to take place instantaneously at a point between the invocation and response of the operation.

To show the property of equivalence to a single-overlay system, we examine the steps of the protocols to insert, remove, and copy data between overlays, and we show that each of these steps always cause a concurrent write or a read operation to occur at a proper overlay: a write always occurs at an overlay that is not occluded (at the time the write is applied to the overlay), so that the write behaves equivalently as in the single-overlay system; and a read always occurs at the highest overlay with data. The proof requires an exhaustive examination of all cases, which is long but conceptually simple.

Availability and fault tolerance of migration. We optimize to provide high-availability for reads and writes; migration operations may pause due to failures. A coordinator crash affects only its own migration operations: we designed the protocols so all clients continue reading and writing consistently without blocking if the coordinator crashes. We recover from coordinator crashes using standard techniques. The coordinator logs each operation and each step within the operation; the log is stored in Nomad itself. If the coordinator crashes, another coordinator reads the log and picks up from where the crashed coordinator left off.

Moving the coordinator. There is a unique coordinator per container, indicated by the directory service, but the coordinator can be easily changed, as follows. The old coordinator finishes its current operation and then performs three actions: (1) start the new coordinator, (2) change the pointer at the directory service, (3) stop.

4.6 Replication

Data replication can be implemented at two places in the component stack: at the storage node level, called *node-level replication*, or at the directory level, called *directory-level replication*.

With node-level replication, a storage node is responsible for replicating itself, and all the replicas are treated by the higher layers as a single virtual node. The migration engine is above the replication mechanism, and we migrate data from one virtual node to another as if the node were not replicated at all. For example, if there are two replicas r1 and r2 of a storage node, they are both treated as virtual node r; containers in r can be migrated to another virtual node s that could have replicas s1 and s2. The advantage of node-level replication is that it is extremely simple and modular, because migration is decoupled from replication. For example, node-level repli-

[3] This holds when overlays are not replicated. Replication is discussed in Section 4.6. It provides a consistency guarantee that is dictated by the replication scheme; for instance, asynchronous replication provides only eventual consistency.

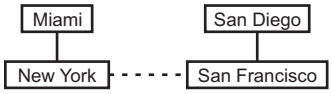


Figure 6: Replication of base overlays. The overlays in New York and San Francisco are asynchronous replicas. There are overlays in Miami and San Diego used as caches of these locations.

cation can be easily provided using a disk array at the storage nodes, or by using state machines coordinated via Paxos [24]. The drawback of node-level replication is that it cannot benefit from the versatility of overlays—for instance, we cannot use overlays to cache or migrate individual replicas, since these replicas are abstracted at a low level in the system.

With directory-level replication, the directory service maps an object/container to several servers (instead of a single server) holding replicas of the base overlay; these replicas are coordinated by the read-write protocol used by clients. The migration engine is below the replication mechanism, and migration moves data from one physical storage server to another. With directory-level replication, individual replicas can benefit from overlays, as illustrated in Figure 6. This scheme is particularly useful when data is replicated across data centers. We now explain how Nomad can be extended to support replication of this sort. (This extension is not implemented in our prototype.) The replicated base overlays are established when a client creates a container and indicates that it should be replicated. Each replicated base overlay may subsequently have a different stack of overlays on top of it, so the stacks are not copies of each other. The data in the different overlay stacks are kept in sync using the desired replication scheme. We believe overlays should work with most replication schemes, by treating each overlay stack as a black-box to which the desired replication protocol issues writes and reads. We illustrate how this is done via three well-known replication schemes—asynchronous primary-backup, asynchronous timestamped, and synchronous. Under all schemes, the directory service indicates the locations of all replicas of the base overlay; when a client writes to one of the overlay stacks, it performs the write at the other stacks as well; to write on a given stack, the client uses the write procedure described in Section 4.3. We now explain the specifics of each replication scheme.

With asynchronous primary-backup replication, one of the overlay stacks is designated as the primary and the other stacks are read-only; writes are only permitted at the primary stack, and the client applies the write asynchronously (in the background) to the other stacks.

With asynchronous timestamped replication, writes are permitted at all replica stacks, and the clients apply the writes asynchronously to the other stacks; a write includes a unique real-time timestamp to order concurrent writes by other clients at other replicas. This is a standard technique: when writes occur at different replicas, the write with higher timestamp obliterates the other writes; if a replica receives a write with a lower timestamp than the data it has, the replica ignores the write. Note that timestamps are globally unique (done by appending a machine identifier to break ties). To obtain timestamps, we assume that clocks are synchronized, say via NTP; machines with faulty clocks can be disabled using a simple monitoring service. Timestamps are kept forever for each extent. We believe that is a small overhead, but if desired it is possible to garbage collect the timestamp at a replica after it is known that the data at other replicas cannot have a smaller timestamp, using the convention that data with no timestamp is treated as having a $-\infty$ timestamp.

With both schemes above (asynchronous primary-backup or timestamped), if a client fails while writing, the write may be applied to some but not all replicas. For that reason, the migration coordinator runs a cleaner that periodically checks for these failed writes and completes them. To make it easy to recognize the failed writes, the client leaves a mark in the overlays that it writes to, which the client clears asynchronously after the client has written to all replicas. If the client crashes without having written to all replicas, the marker will be left at the overlay. Both asynchronous schemes described above provide eventual consistency.

With synchronous replication, when a client issues a write to one of the overlay stacks, the client must write to the other replica stacks synchronously (i.e., before the write is acknowledged to the client). As with asynchronous replication, a write includes a timestamp to order concurrent writes, and we use a marker to recognize failed writes. To read, a client reads from one of the overlay stacks and then checks that the data being read has no marker (the common case); if it has a marker, the client writes the data and its timestamp synchronously to the other replica stacks. This is done to ensure that later reads at other replicas cannot not return data that is older than the data being returned by the current read, to provide a strong form of consistency. This synchronous replication scheme provides linearizability [20].

With all of the replication schemes, we can migrate a replica using the procedure described in Section 4.4.

4.7 Multi-way caching and split overlays

In Section 4.4, we described how to use overlays to cache data at one location. It may be desirable to set up multi-way caches, where data is cached at many locations from a single replica. In other words, there is a single replica of the full data set, and many caches each with some (possibly overlapping) part of the data set. To do this, we need the notion of a split overlay, which is illustrated in Figure 7. (This extension is not implemented in our prototype.)

Caches exist for performance, and so they should al-

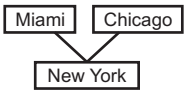


Figure 7: Split overlays are used to cache the data in New York at both San Francisco and Chicago.

low for efficient reads and writes without synchronization across the caches in different data centers. As a result, split overlays provide only a weak form of consistency, namely eventual consistency. A split overlay occurs when an overlay has several overlays over it on the next level; each of these overlays is called a *split*. A split is established using the operation to insert a new overlay of Figure 5, with a special flag indicating it is a split. This flag causes the overlay below the point of insertion to store an additional *above-pointer* to the new overlay. When a write occurs at one of the splits, the write occurs as in any other overlay: the data is marked as unique (dirty) and it is *not* propagated to other splits. A client can cause the data at a split to be copied to the common overlay below, via the *start_copy* operation of Figure 5. This corresponds to flushing the cache. When the overlay below receives the data, it invalidates older data at the same position in the other splits, using the data's timestamps to decide what is older. As a result, content initially written to a split is not visible at the other splits, but as soon as the content is flushed down, it becomes visible. In the example of Figure 7, when the dirty writes in Miami are flushed to the common overlay in New York, New York sends an invalidation message to Chicago, which causes Chicago to discard any older writes. Subsequent reads in Chicago will read the data from New York.

In general, the protocol works as follows. Suppose there is an overlay O at level k and m splits $O_1, \ldots, O_m$ at level $k+1$. If a write occurs at an overlay O_i or above, the write remains in the split with the unique bit set. Subsequently, when the data at O_i is copied to overlay O, the server of overlay O sends an invalidation message with the data's timestamp and position to the other splits at level $k+1$. Each of these overlays checks whether it has older data in the same position and, if it does, removes such data from the overlay. If there are further overlays above, the invalidation message is forwarded recursively. If the server of an overlay crashes and recovers, it may lose this invalidation message (e.g., it may have received the message and then crashed without having the time to process it). For that reason, the overlay that originates the message retransmits it periodically until it gets acknowledgements from the top overlays in each branch. An overlay may process the same invalidation message twice, but this is not a problem since the message is idempotent.

Now suppose that we want to remove a split overlay—say, in the example of Figure 7, we wish to remove the split in Miami and be left with an unsplit stack with New

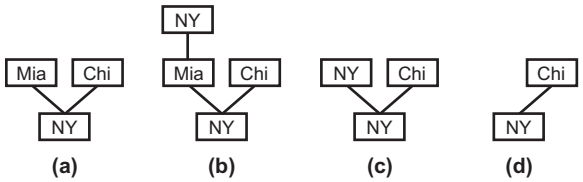


Figure 8: Removing a split overlay. Miami (Mia) and Chicago (Chi) are initially split and we wish to remove Miami.

York and Chicago. We use the procedure to remove an overlay described in Section 4.5, with a small modification to incorporate the invalidation mechanism that we described above. More precisely, as shown in Figure 8, we first create an overlay in New York on top of Miami; we then copy the data from Miami to the overlay above it in New York; next we remove the overlay in Miami. We are left with a split overlay where New York is on top of New York, as shown in Figure 8(c). The final step is to locally copy the data from the higher to the lower overlay in New York, send invalidation messages to Chicago, and finally remove the higher overlay in New York.

Note that split overlays can be migrated as well, using the procedure described in Section 4.4. This modularity makes overlays a flexible mechanism.

5 The policy of geo-distribution

Thus far, we have described the Nomad system and the *mechanism* of migration it provides. We now discuss the *policy* of migration: what data to migrate, where to migrate it, and when to do so. There is no one-size-fits-all policy: migration policies depend on the specifics of data center deployments as well as application requirements. Below, we describe some of the key deployment factors that a policy layer must take into consideration.

Data center granularity: A geo-distributed system may consist of a few large data centers, or many small data centers. The former characterizes current deployments of large companies such as Microsoft, while the latter alternative is based on the use of containers [12]. Smaller data centers allow data to be closer to users, but place greater strain on the migration scheme.

Network costs: A geo-distributed system usually communicates on two different networks: an internal one between data centers, and an external one to connect with users (the Internet). The cost model for the internal network can vary. If the internal network consists of dedicated, privately owned links, the cost and speed of the network are fixed. Alternatively, network cost on leased links can depend on the amount of data transferred; for example, it is common for network operators to bill customers based on 95th percentile network utilization. The external network is provided by Internet ISPs, and the cost depends on the amount of data transferred in and out.

Access protocols: When a user accesses the service via the web, the request is redirected via DNS-based load-

balancing to a local data center. If the data the user needs (e.g., her inbox) is in a different data center, the system has two options:

- *Redirect*. The system redirects the user to the appropriate data center. Subsequently, the local data center is not in the communication path, and the communication from the user to the appropriate data center is via the Internet. This option saves bandwidth on the internal network, but may impair the user experience because the Internet provides no quality of service.
- *Relay.* The local data center continues to serve the user and fetches needed data from the remote data center using the internal network. Thus, the local data center is in the communication path. This option tends to provide more predictable access times, and it allows the local data center to satisfy parts of the request locally (e.g., ads). However, this option is more expensive because one must provision the internal network adequately.

Against this backdrop, a migration policy must trade off migration bandwidth on the internal network for reduced access latencies. If the system uses the *Relay* option, the policy also has to factor in the bandwidth cost on the internal network of remote accesses on non-migrated data; in the *Redirect* model, this is not a factor.

A policy layer for online services. In addition to the deployment factors listed above, migration policies also depend on application characteristics. For example, Nomad could be used with a policy layer that periodically computes optimal placements for data given the location of recent accesses of users and the capacity of each data center, as in the Volley system [7]. The effectiveness of this policy depends on the application; it works well if user movements tend to be permanent, but can result in excessive migration if users move back and forth.

We describe a new policy layer based on predictions of future user movement. This layer provides insight into how predictions of user movement can be used to achieve access locality while eliminating unnecessary migrations.

Our policy layer makes the decision to migrate a user based on the cost of doing so versus its predicted future benefit. If we could perfectly predict the benefit, this choice would be easy; since we cannot, we must settle for heuristics that use past behavior to try to predict future accesses at the same location. We consider three simple migration policies. They all monitor the location of the user when she accesses the data, and trigger migration when a condition is met. The three conditions we consider are the following:

- *Count:* Data is accessed from the same remote location a certain number of times (e.g., 10 times);
- *Time:* Data is accessed from the same remote location for a certain period (e.g., 10 days);
- *Rate:* Data is accessed from the same remote location above a certain rate (e.g., 3 accesses per day).

For example, suppose Alice moves from Redmond to London. Suppose she accesses her mailbox twice on each of the first five days in London, twelve times on the sixth day, and then returns to Redmond on the seventh day. The Count-based policy with a threshold of 10 accesses migrates her mailbox to London on the fifth day; the Time-based policy with a threshold of 10 days does not migrate her mailbox. The Rate-based policy with a threshold rate of 3 accesses per day migrates her mailbox to London on the sixth day. In this case, the Time-based policy is the best. Since Alice returns to Redmond after a short trip, her mailbox should not be migrated. In other cases, the Count and Rate-based policies may work better.

We later report on the efficacy of these different policies when applied to real user traces taken from a large web mail service. Since these policies are predicated on the movement of users in the real world (rather than the semantics of a specific application like webmail), we believe the results to be relevant for other web applications, such as the ones mentioned in Section 2.

6 Implementation

We implemented overlays in a prototype of Nomad as we described in Section 4, except that we did not implement replication (Section 4.6) and split overlays (Section 4.7)—which are unnecessary to compare Nomad to other migration schemes. The Nomad prototype has 6,000 lines of C# code, comprising a client library, a storage server, and directory server. The directory server provides RPCs to get and set the location of the base overlay of a container given its 64-bit identifier. A storage server provides RPCs for the following: (1) Read/write to an overlay; (2) Get the overlay above and below of an overlay; (3) Delete an overlay; (4) Create new top overlay at another storage server for a given overlay; (5) Copy data of an overlay to its upper overlay; (6) Migrate an overlay to another storage server.

Storage servers store data for an overlay as a directory in the local file system, containing a metadata file and one file for each extent of the overlay, named by the object id, start offset and end offset. A write to an overlay may merge extent files. Storage servers cache overlay metadata in memory to improve read performance.

7 Evaluation of mechanism

In this section, we evaluate the use of overlays for migration, through experiments that measure overlay overheads, verify their flexibility, and compare their performance against alternatives.

7.1 Alternative schemes for migration

We consider two alternative schemes for migration, which are often used in practice:

- *Lock-based migration:* While the data is copied from the old location to the new location, the system blocks write operations. Read operations are not blocked; they are served at the old location. Writes are unblocked after the old location is marked as invalid and the directory is updated to point to the new location.
- *Log-based migration:* The system creates a log at the old location to store the updates while the data is copied from the old to the new location. During the copying, reads and writes are served at the old location. Once the copying is finished, the system blocks write operations, copies the log from the old to the new location, marks the old location as invalid, modifies the directory to point to the new location, and then unblocks write operations.

7.2 Experimental setup

Our setup consists of machines in data centers in five locations: Mountain View (CA), Redmond (WA), Boston (MA), Cambridge (UK), and Beijing (CN). Each machine consists of a PC with two quad-core 2.27 GHz Xeon processors, 16 GB of RAM, an internal disk array with several 10,000 rpm SAS disks, running 64-bit Windows Server 2008 R2. Machines are connected to a Gigabit switch, and the various locations are connected by a dedicated network. The median ping latencies between locations are as follows, in ms:

	WA	MA	CN	UK
CA	19	112	167	237
WA		79	141	204
MA			220	283
CN				345

7.3 Overhead of overlays

We now evaluate the overhead imposed by overlays.

Access latency. In this experiment, we measure the latency that overlays incur on accesses to data. A client reads or writes a small object with up to five overlays in different locations, as we measure the latency of reads or writes in two separate experiments. The client is in the same location as the top overlay, which is typical of having a cache at the local data center.

Figure 9 shows the results for writes. We see that the first write incurs a higher latency, because the client needs to traverse overlays in different locations from bottom to top. Once the client learns the top overlay, it caches it for the entire container; subsequent writes on any object of the container are much faster, incurring only a local-area-network latency plus a disk-write latency. We can avoid the higher latency for the first write by keeping a hint of the location of the highest overlay at the directory service. We implemented this optimization, but Figure 9 shows the unoptimized scheme, representative of the worst case.

For reads, the situation is similar (not shown). The first read discovers the overlays, while subsequent reads are faster. The exact latency for subsequent reads depends on the first overlay that has the data: if it is the top overlay, it is just a local-area-network latency, otherwise it is the sum of the latencies to communicate with each successive overlay until data is found. We implemented an optimization so that if a read does not hit the top overlay then it writes the data there so that the next read of the same data will be faster—thereby treating the top overlay as a cache. A client-settable flag determines whether this optimization is enabled or not.

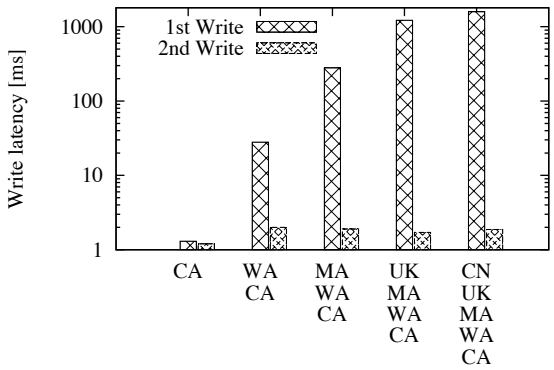


Figure 9: Write latency using several overlays in many geographical locations. The x-axis indicates the number and location of overlays.

Overlay space. We measure the space overhead of overlays. The on-disk or in-memory metadata for each overlay is smaller than 1 KB. A larger overhead occurs because lower overlays may store useless data occluded by higher overlays. In theory, a container's storage space across all servers could be multiplied by the number of overlays. In practice, most overlays are usually empty, but even if they were not, it is easy to introduce a garbage collection mechanism that periodically detects and erases occluded data (we have not implemented this). The garbage collection period can be many minutes because most storage systems are over-provisioned.

7.4 Flexibility of migration mechanism

In terms of functionality, overlays provide the flexibility to migrate data while clients are concurrently reading and write data; during migration, the system may create or flush a remote cache, and the cache itself could be independently migrated. Lock-based and log-based migration do not provide this flexibility, but they could support a static cache layer, which cannot be removed or added. With extensions, lock-based and log-based migration could support migration of the cache layer, possibly concurrently with migration of the storage layer, but this requires additional careful design. The flexibility of each scheme is summarized in the table below.

Scheme	support static cache	create cache layer	remove cache layer	migrate cache
Lock-based	Yes	No	No	Extension
Log-based	Yes	No	No	Extension
Overlay	Yes	Yes	Yes	Yes

We devised an experiment to demonstrate the flexibility of overlays as a user moves between four locations. The user is initially at the United Kingdom (UK), where she has 50 MB of data. Her workload consists of reading or writing small objects within some working set of size 2 MB. At time 60s, she moves to Boston (MA). At that time, we start a process to migrate her data to MA, starting with her active set, using 300 Kbps of bandwidth. This could be the maximum bandwidth a given user is allowed to consume in a system with many users. The entire migration will take around 1600s, but her active set can be copied in 60s. At time 180s she moves to Redmond (WA), but the migration UK-MA has not finished yet, so we create a cache in the WA data center and start populating it with her working set, copying the data from the MA data center at a rate of 400 Kbps. At time 300s, she moves to California (CA), and we migrate her cache from WA to CA at a rate of 400 Kbps. There are separate experiments for reads and writes. Note that in this experiment, we compress travel time so that we can fit the scenario in one small graph. In a more realistic setting, the user may remain at a location for several days, and the corresponding graph would look like the one we give, except it would have large segments depicting no interesting information while the user remains at a location.

Figure 10 shows the latency of the user's reads and writes during this scenario. The latency refers to a client running on the user's behalf at the data center closest to the user—this client could be a web application that reads and writes within the data center. We can see that initially the latency is close to 0, reflecting a local access. At

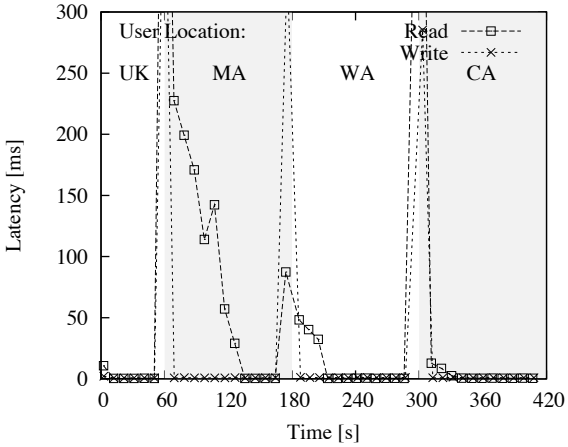


Figure 10: Read and write latency as the user moves between 4 locations and we migrate her data and cached data as she moves.

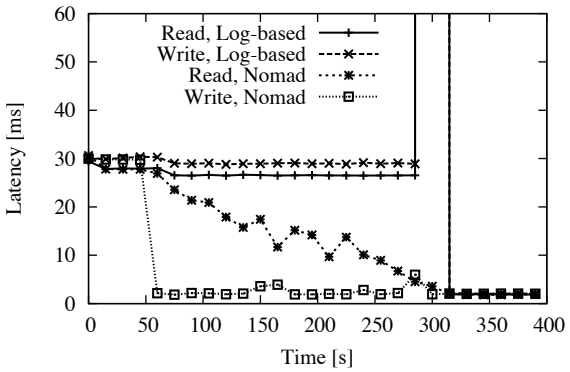


Figure 11: Read and write latency before, during, and after migration. Migration starts at 50s and completes at around 300s.

60s, once the user moves and migration starts, the latency spikes to around 1s for a few accesses: this is the time it takes to lookup the user's data and traverse the overlay stack. Soon after, the write latency drops to ≈0, because writes are now done locally. The read latency gradually drops as the working set is migrated from UK to MA, which takes ≈70s. At 180s and 300s, we observe the same phenomenon, except that the working set is copied faster, in ≈40s, since more bandwidth is available for populating or migrating the client's cache.

7.5 Performance comparison

In this experiment, we evaluate the latency of accesses to data during migration. A container with 50 MB of data is initially located in the WA data center (source location) and a client periodically reads or writes small objects in that container from the CA data center. At 50s, the system starts migrating the container to CA (destination location), which is the same data center as the client, using 2 Mbps of bandwidth. We measure the latency of reads and writes to the objects as the migration progresses.

Figure 11 shows the results for Nomad and log-based migration. With the latter, there is a period of write unavailability at the end of migration when the log is copied. The write unavailability is given by the formula:

$$\frac{\textit{filesize}}{\textit{migrate-rate}} \times \frac{\textit{write-rate}}{\textit{migrate-rate}} = \frac{\textit{filesize} \times \textit{write-rate}}{\textit{migrate-rate}^2}$$

where *write-rate* refers to the new writes during migration, and *migrate-rate* is the rate at which data is copied.

The unavailability can be reduced by using a second log to store updates while the log is being migrated (and this can be done repeatedly).

Log-based migration has two other drawbacks compared to Nomad. First, read and write latency remains high during migration because operations are served at the source location until migration is completed. In contrast, with Nomad, the write latency immediately decreases when the migration starts while the read latency progressively decreases, because the client reads ran-

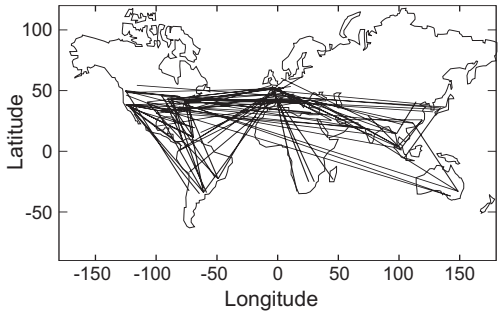


Figure 12: Movement patterns of sample users that traveled >2000 miles for >3 weeks with >7 accesses. Disclaimer: sample is not statistically significant, provided for illustrative purposes only, not necessarily representative of Hotmail's market penetration.

domly chosen objects and, as time passes, a larger fraction of these objects are in the same location as the client. Another drawback of log-based migration is that it consumes three times as much bandwidth for new writes done during migration (not shown on the graph). These writes must be (1) received at the source, (2) sent from the source, and (3) received at the destination. Intuitively, this is because writes during migration are sent to the source. This must be so, because reads are served at the source. In contrast, with overlay-based migration, writes can go directly to the destination, because the system can use overlays to serve reads from a combination of the source and the destination.

We also tried lock-based migration (not shown on the graph). The result is what one would expect: during migration there are no writes, and reads have high latency since they are served at the source location.

8 Evaluation of policy

We evaluate three simple migration policies using real traces from Hotmail. Generally, migration can be triggered by a combination for factors, including balancing of storage capacity, balancing of bandwidth, and movement of users. The policies we consider here are based on movement of users; a more comprehensive set of policies may consider the other factors as well [7]. We evaluate policy independent of the mechanism used for migration, to separate concerns. Our traces comprise the login records of ≈50,000 randomly chosen Hotmail users, collected over two months (Aug-Sept 2009). For each user, it contains the login time and IP address from which the user logged in. We use a public IP-based geolocation service to map each IP address to latitude, longitude coordinates. To apply our policies, we view each login as a separate access, and the unit of migration is a mailbox. Figure 12 shows examples of the movement patterns in the trace.

To eliminate errors introduced by the geo-location service, we first pre-process the trace by clustering sequences of close-by accesses by a user (less than 150 miles from each other) into *visits*. Thus, if a user logged in twice from New York City and twice from New Jersey (which are very close), we consider that as a single visit of four accesses. If the user then logs in from Seattle, and later again from New York City, that is three visits.

As we explained in Section 5, data center granularity is an important consideration: the movement of a user is only relevant for migration if the closest data center to the user changes. We consider that the data center changes only if the distance between one visit and the next is above a threshold. We consider three such thresholds, corresponding to three data center granularities:

- *Large-DC:* Threshold is *2000* miles, corresponding to a deployment with massive data centers serving a large area. 1% of the users in the trace have visits that satisfy this criteria.
- *Medium-DC:* Threshold is *1000* miles, corresponding to data centers serving a mid-sized geographical region. 1.8% of the users in the trace have visits that satisfy this.
- *Small-DC:* Threshold is *450* miles, corresponding to having data centers for individual states or metropolitan areas. 3.5% of the users in the trace have visits that satisfy this.

For each data center granularity, we study the three migration policies described in Section 5. For each user, we scan the trace until we find a *remote* visit—a visit whose distance from the first visit exceeds the distance threshold (2000, 1000 or 450). We then apply the policy to that remote visit to see if migration is triggered; for example, the Count policy with a threshold of 10 triggers migration if the visit contains 10 or more accesses.

Figure 13 shows what fraction of users trigger migration as a function of each policy's threshold. The fraction is relative to the users with at least one remote visit.

We now examine the effectiveness of the three policies using the metric of *saved remote accesses*, which measures the benefit of migration: these are accesses that, without migration, would have been served at the original data server far from the user, but with migration, are served from a data center close to the user. For example, if a user accesses her mailbox 500 times during a trip, and we use the Count policy with a threshold of 50, the number of saved accesses is 450.

Figure 14 shows the average number of saved accesses per migrated mailbox on the y-axis. Each point corresponds to a different threshold for each policy, for the Large-DC and Small-DC granularity (the Medium-DC is between those two, and not shown). The x-axis has the percentage of migrated users using that threshold; points to the right correspond to thresholds that migrate more users. For the Count and Rate policies, the curves decrease monotonically as more users are migrated; therefore, most of the migration benefit is obtained by choos-

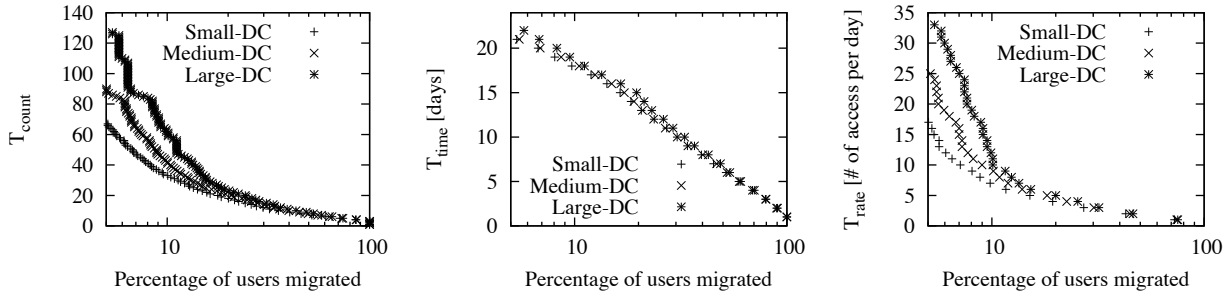


Figure 13: Effect of varying thresholds for Count (Left), Time (Middle) and Rate (Right) policies.

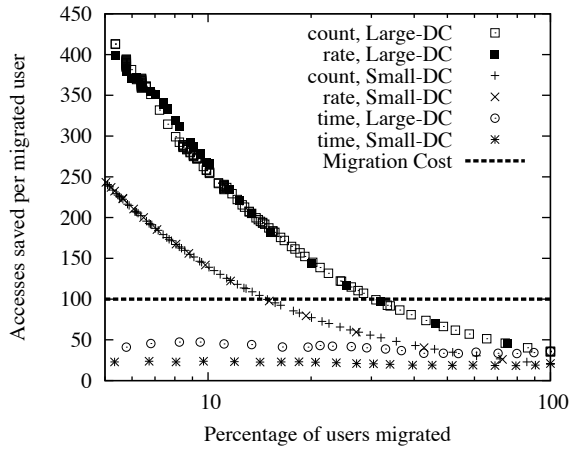


Figure 14: The benefit per migration for different policies.

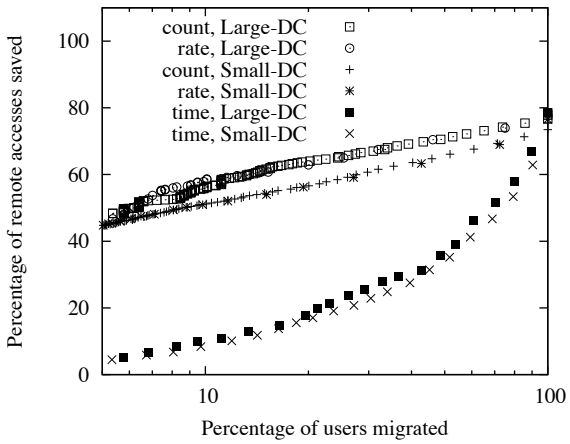


Figure 16: Effectiveness of policies in saving remote accesses.

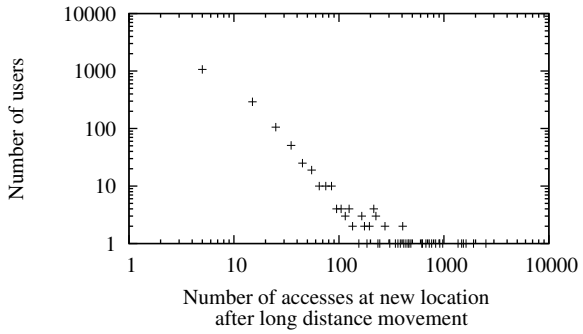


Figure 15: Distribution of # of accesses during remote visits.

ing a threshold that migrates few users, with more aggressive thresholds providing diminishing returns. The graph shows that the Count and Rate-based policies are better than the Time-based policy.

Figure 14 can be used to determine the break-even point for each policy, where the benefit of migration (saved remote accesses) outweighs the cost of migration, assuming the *Relay* access model (Section 5), in which remote accesses and migration consume bandwidth on the same internal network and can be quantitatively compared. The line marked "Migration Cost" illustrates a specific case where the bandwidth needed for migration is equivalent to 100 remote accesses (e.g., for migrating recent emails). This number could be different, corresponding to different horizontal lines. The break-even point is where the horizontal line intersects the curve of each policy.

In both the *Relay* and the *Redirect* model, the internal network may have limited bandwidth for migration if other traffic is given priority. In this case, migration can be done opportunistically, using spare bandwidth during intermittent idleness of the links. Which mailboxes should be migrated to offer the greatest benefit? Figure 14 indicates that mailboxes migrated by the largest thresholds offer more benefit than those migrated with smaller thresholds. This argues for adaptively changing the threshold to match available bandwidth.

Figure 15 explains why the Count policy works well. It plots the distribution of the number of accesses per remote visit; we see that the distribution is linear on a log-log scale and can be fitted to a heavy-tailed Pareto distribution, with a few visits containing many accesses. This explains the monotonically decreasing benefit of the Count policy on the previous graph: it can be analytically shown that a Pareto distribution always exhibits this property (we omit the analysis for lack of space).

Finally, we determine the overall effectiveness of the different policies by measuring the total percentage of remote accesses saved. Figure 16 plots this metric on the y-axis. The Count and Rate policies are very effective in saving remote accesses; for example, with thresholds that migrate 10% of all users, both policies save 55 to 60% of all remote accesses in the Large-DC case. As expected, the Time policy is not very effective, requiring almost all users to be migrated to achieve similar savings.

9 Related work

Migration mechanism. There has been a lot of work on distributed file systems [15, 17, 18, 21, 23, 28, 32–34]. These systems either do not support migration, or employ lock-based or log-based migration. For example, AFS [21] allows a volume to be moved from one server to another using log-based migration. xFS [33] allows a client to borrow a file for exclusive writing, but this is different from migration since the file is ultimately returned to its home server, which serves as a coordination point (e.g., if multiple clients want to write). In Pangea [32], migration is achieved by simply creating a new replica, but the system provides only eventual consistency, in contrast to Nomad. Ceph [34] allows (the metadata of) a directory to be moved from one server to another, using lock-based migration. Coda [28] allows clients to hoard files for disconnected operation; this is different from migration since hoarded files are eventually returned to the server that owns the file. Farsite [17] appears to support migration of metadata, by changing the mapping from identifier prefixes to servers, using a lock to avoid races. GFS [18] appears to support migration of chunks, by copying a chunk from one server to another, and then updating the mapping from chunk id's to servers at the master using a lock to avoid races.

Migration of a virtual machine (VM) is a well understood technology, done by VMware [3], and a couple of years later in Xen [13]. This technology is about moving a functional VM to another host. In a first round, the entire VM's memory is copied; if a page of memory changes after being copied, it is marked dirty and the marked pages are copied in a subsequent round. The system may execute many rounds as further pages are marked, until it decides to pause the VM, copy the remaining dirty pages, and start the destination VM. Subsequent work on VM migration considered the copying of direct attached storage [22]. This body of work is different from ours because it focuses on migration of data accessed by a single machine whether in memory or disk, whereas we consider a *distributed* setting and must address the required coordination among several servers (which we do via overlays).

In PNUTS [14], data is replicated across data centers and migration consists of changing the master replica. This scheme requires many replicas across data centers, which we must avoid. Cloud storage services support migration between different locations. In Amazon's storage server [4], the approach to migrate an elastic block store (EBS) is as follows: (1) the user stores a snapshot of the EBS in S3, and (2) the user creates an EBS at a different location and populates it with the content in S3. This scheme, though simple, will fail to migrate any writes done on the original EBS during migration.

Distributed object systems support migration of objects (see [11, Section 5.2.2]), which is more complex than migrating data, since objects have threads, TCP connections, and other contextual state. The migration mechanism employed is lock-based migration.

Commercial disk array solutions such as the HP-UX logical volume manager [26] support online migration by essentially using the logging technique. In this context, Aqueduct [25] is a system that controls migration traffic to maintain low access latencies during migration.

The work in [8, 19, 27, 30] shows how to add or remove replicas in a replicated state machine or a quorum system. These techniques can be used for migration, by adding a replica at a new location and removing from the old location. This work is theoretical and would be inefficient for wide-area-network storage.

Migration policy. Volley [7] uses system logs of accesses to determine placement of data across data centers, based on data access interdependencies, who has accessed the data and when, and a balance of storage capacity across data centers. This is different from our work because of four reasons: (1) Volley's placement algorithm computes a global placement for *all* data, whereas our scheme determines where a particular piece of data should be migrated, (2) Volley's algorithm does not consider the cost of migrating data, so the algorithm is not applicable to our setting where migration *has* as a cost; in fact, the consideration of cost-benefit of migration is central to our scheme, (3) Volley does not propose mechanisms for migration, (4) Volley does not attempt to predict future user movement.

Previously, data placement has been extensively studied in the context of web servers and Content Delivery Networks (CDNs) [29]. Since data in these settings is read-only, most of these solutions are centered on replica creation and placement.

Predicting the movement of users has been explored in mobile systems [10]. In contrast to this work, we are concerned with predicting movement at coarse grain (e.g., is user staying in Asia or returning to Europe?) instead of precise locations.

10 Conclusion

This paper addresses the problem of providing online migration of data across data centers—a problem that occurs as users move and/or data centers become unbal-

anced due to new applications, unforeseen growth, and new data centers. To design a migratable storage system, we propose an abstraction called distributed data overlays, which has a simple real-world analogy based on transparent pieces of paper. We implemented this abstraction within a prototype of a key-value object store called Nomad, which spans multiple data centers and allows for migration and caching of object containers across data centers. It is very easy to use overlays to implement migration; the complexity is hidden by the protocols that implement overlays (which we provide), as these protocols must coordinate concurrent reads, writes, migrations, and the dynamic creation and removal of remote caches. We also study some policies that might trigger the migration mechanism based on user movement, but other policies could be applied as well [7].

Acknowledgements. We are grateful to Asim Kadav, Jean-Philippe Martin, Amar Phanishayee, and Fang Yu for helpful comments on an earlier draft. We are also grateful to our shepherd Wilson Hsieh for having provided many useful comments throughout the paper.

References

[1] `http://www.verizonbusiness.com/terms/us/products/internet/leasedline/`.

[2] `http://www.swisscom.ch/solutions/Resources-en/Dokumente/factsheet/00276-factsheet-private-line-international-en`.

[3] VMware news release, VirtualCenter, Nov. 2003. `http://www.vmware.com/company/news/releases/virtualcenter.html`.

[4] Amazon elastic block store, Jan. 2011. `http://aws.amazon.com/ebs`.

[5] Amazon simple storage service, Jan. 2011. `http://aws.amazon.com/s3`.

[6] Microsoft azure blog storage, Jan. 2011. `http://msdn.microsoft.com/en-us/library/dd135733.aspx`.

[7] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan. Volley: Automated data placement for geo-distributed cloud services. In *Symposium on Networked Systems Design and Implementation*, pages 17–32, Apr. 2010.

[8] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58(2), Apr. 2011.

[9] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3):5:1–5:48, Nov. 2009.

[10] D. Ashbrook and T. Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing*, 7(5):275–286, Oct. 2003.

[11] R. S. Chin and S. T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, Mar. 1991.

[12] K. Church, A. Greenberg, and J. Hamilton. On delivering embarrassingly distributed cloud services. In *ACM Hot Topics in Networks Workshop*, pages 55–60, Oct. 2008.

[13] C. Clark et al. Live migration of virtual machines. In *Symposium on Networked Systems Design and Implementation*, pages 273–286, May 2005.

[14] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases*, pages 1277–1288, Aug. 2008.

[15] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *ACM Symposium on Operating Systems Principles*, pages 202–215, Oct. 2001.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.

[17] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 321–334, Nov. 2006.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[19] S. Gilbert, N. A. Lynch, and A. A. Shvartsman. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. *Distributed Computing*, 23(4):225–272, Dec. 2010.

[20] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[21] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.

[22] A. Kadav and M. M. Swift. Live migration of direct-access devices. *ACM SIGOPS Operating Systems Review*, 43(3):95–104, July 2009.

[23] M. Kim, L. P. Cox, and B. D. Noble. Safety, visibility, and performance in a wide-area file system. In *USENIX Conference on File and Storage Technologies*, pages 131–144, Jan. 2002.

[24] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[25] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *USENIX Conference on File and Storage Technologies*, pages 219–230, Jan. 2002.

[26] T. Madell. *Disk and file management tasks on HP-UX*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

[27] J.-P. Martin and L. Alvisi. A framework for dynamic byzantine storage. In *International Conference on Dependable Systems and Networks*, pages 325–334, June 2004.

[28] L. B. Mummert, M. R. Eblig, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *ACM Symposium on Operating Systems Principles*, pages 143–155, Dec. 1995.

[29] L. Qiu, V. N. Padmanabhan, and G. M. Voelker. On the placement of web server replicas. In *IEEE International Conference on Computer Communications*, pages 1587–1596, Apr. 2001.

[30] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report TR/932, MIT LCS, Dec. 2003.

[31] Y. Saito, S. Frolund, A. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *International conference on Architectural support for programming languages and operating systems*, pages 48–58, Oct. 2004.

[32] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mhalingam. Taming aggressive replication in the Pangaea wide-area file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 15–30, Dec. 2002.

[33] R. Y. Wang and T. E. Anderson. xFS: A wide area mass storage file system. In *Workshop on Workstation Operating Systems*, pages 71–78, Oct. 1993.

[34] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 307–320, Nov. 2006.

Slow Down or Sleep, that is the Question

Etienne Le Sueur and Gernot Heiser
NICTA and The University of New South Wales
{elesueur,gernot}@nicta.com.au

Abstract

Energy consumption has become a major concern for all computing systems, from servers in data-centres to mobile phones. Processor manufacturers have reacted to this by implementing power-management mechanisms in the hardware and researchers have investigated how operating systems can make use of those mechanisms to minimise energy consumption. Much of this research has focused on a single class of systems and compute-intensive workloads.

Missing is an examination of how much energy can actually be saved when running realistic workloads on different classes of systems. This paper compares the effects of using dynamic voltage and frequency scaling (DVFS) and sleep states on platforms using server, desktop and embedded processors. It also analyses workloads that represent real-world uses of those systems. In these circumstances, we find that usage of power-management mechanisms is not clear-cut, and that it is critical to analyse the system as a whole, including the workload, to determine whether using mechanisms such as DVFS will be effective at reducing energy consumption.

1 Introduction

Energy consumption, which was previously only a concern for mobile systems, has become important for all classes of systems. Processor manufacturers have implemented various mechanisms for managing energy consumption, some of which allow the operating system (OS) to trade performance against power. These are collectively known as *power-management* mechanisms.

Some mechanisms, such as dynamic voltage and frequency scaling (DVFS) and fine-grained clock-gating, are used while the processor is executing instructions, while other mechanisms are designed to reduce power consumption when the processor is not in use. The most common of these is the *sleep mode* or C state, which can be used to put the processor (or some part of it) into a low-power mode. Modern processors include several different C states which result in different power consumption and have different overheads.

These two types of power-management mechanisms present a trade-off to the OS designer: either run a task at a reduced CPU frequency (consuming less power) but remain active for a longer period of time (i.e. *slow down*), or, run a task at a high CPU frequency (with higher power draw) and as soon as possible enter a low-power idle state (an approach known as *race-to-halt* or *sleep*).

Prior research has focused on improving energy efficiency by using DVFS, resulting in a number of techniques that can be employed by the OS [6, 8, 10]. However, much of this research used workloads, especially SPEC CPU benchmarks, which are not representative of real-world system use. In addition, the methodology used in some of these studies unfairly biases the results toward those using high CPU frequencies—less static energy is consumed at a high frequency due to reduced task execution time. Consequently, the findings from this research must be interpreted with caution.

We recently presented an analysis of several server-class systems examining the effectiveness of DVFS at improving the energy efficiency of CPU-intensive workloads, such as the SPEC CPU workloads [5]. However, that work suffered from a common limitation in that it used unrealistic workloads. It also focused exclusively on high-end platforms. In this paper, we address these shortcomings by looking at workloads that are more representative of real system use, such as multimedia decoding/playback and serving of web pages. These workloads exhibit frequent, short idle periods (or bursty behaviour) which allows the trade-offs presented by DVFS and multiple C states to be examined more thoroughly.

We also look at a wider range of platforms in order to better understand the technology trends that we identified in our previous publication. Specifically we look at a desktop-class system based on an Intel Core i7 870 processor (the Dell Vostro 430s) and two platforms built on the most popular high-end low-power processor micro-architectures—the Intel Atom Z550 (the fitPC2) and the ARM Cortex A9-based Texas Instruments OMAP4430 (the Pandaboard). The specifications of these systems are provided in Table 1.

The rest of this paper is structured as follows. Section 2 discusses related work and Section 3 provides an overview of the power-management mechanisms that are available on the three platforms. Section 4 presents our experimental methodology. Section 5 then discusses the findings of our evaluation and Section 6 outlines our conclusions.

System	Dell Vostro 430s	fit-PC2	Pandaboard
Processor	**Intel Core i7 870**	**Intel Atom Z550**	**OMAP 4430 Cortex A9**
ISA	64-bit x86	32-bit x86	32-bit ARMv7
Class	Desktop	Embedded	Embedded
Cores / Threads	4 / 8	1 / 2	2 / 2
Frequency (GHz)	1.2–2.93, TB 3.6	0.8–2.0	0.3–1.008
Voltage (V)	0.65–1.40	0.75–1.1	0.93–1.35
Process	45 nm		
TDP	95 W	2.4 W	Unknown
L2 cache	4×256 KiB	512 KiB	1 MiB (shared)
L3 cache	8 MiB	-	-
Memory	4 GiB DDR3 1,333 MHz	1 GiB DDR2 533 MHz	512 MiB LPDDR2 800 MHz
Storage	500 GiB 3.5" SATA hard-drive	80 GiB SATA 2.5" hard-drive	64 GiB USB SSD 4 GiB SD
MPEG decode assist	None	PowerVR VXD	IVA-HD

Table 1: Specifications of the three processors and systems we analyse [1, 2]. Thermal Design Power (TDP) is the maximum processor power dissipation expected.

2 Related Work

Barroso and Hölzle presented the case for *energy-proportional computing* in 2007 [4]. Their analysis of several thousand servers in a Google data-centre showed that these servers spent most of their time significantly under-utilised. This is an interesting finding, and suggests that power-management research should be focused on workloads that result in this low level of system utilisation. In this paper, we intend to show how DVFS and sleep states can improve energy efficiency for under-utilised systems running real-world workloads, such as serving web pages.

A separate problem which has been tackled by many researchers is when to invoke DVFS and to what level. These decisions are often made by the operating system (OS). OS power management has been under active investigation since 1994 when Weiser et al. introduced the idea of changing the CPU frequency based on the system load [9]. This technique is now widely used, including in mainstream Linux.

More complex systems have been devised which make use of mathematical models to estimate the impact of reduced CPU frequency on the performance of a workload. Systems such as those proposed by Weissel and Bellosa [10] and Snowdon et al. [8] use hardware *performance counters* which are commonly available to parameterise models on which to base power-management decisions. Using these techniques, energy savings of up to 20 % were achieved on systems based processors such as the Pentium-M and PXA255. However, Snowdon et al. focused on using SPEC CPU workloads which do not cover a wide range of real-world use-cases. This limits what can be learnt from this work about the practical potential for energy management.

In 2002, Miyoshi et al. showed that the decrease in slack time resulting from running at a lower CPU frequency could offset any savings achieved by using DVFS [7]. By using a web-server workload similar to our own they found that it was more energy-efficient to run at a high frequency (i.e. *race-to-sleep*) for both high-utilisation and low-utilisation scenarios. However, the systems that were available 10 years ago are very different from those available today. Many low-power idle states are now available, and their usage results in significantly reduced power draw. Furthermore, *static power* is growing as a proportion of total system power, reducing the impact of DVFS on overall power draw.

This paper builds on previous work by looking at a wider range of workload classes and more recent systems with modern power-management mechanisms.

3 CPU Power-management Mechanisms

With the increasing importance of reduced energy consumption, it is not surprising that processor manufacturers have implemented an increasing number of power-management mechanisms. Two of these, DVFS and sleep states are described below.

3.1 DVFS

DVFS is a mechanism that exploits the relationship between the power consumption of a CMOS device, and the frequency at which it is clocked,

$$P = CfV^2 + P_{static}, \tag{1}$$

where C is the sum of capacitances within the circuit (which depends on transistor feature size), f is the operating frequency and V is the supply voltage. P_{static} represents power consumed from leakage mechanisms such as sub-threshold (weak-inversion), short-circuit and gate leakage. The voltage required for stable operation is determined by the frequency at which the circuit is clocked and can be reduced if the frequency is also reduced.

In the past, DVFS has been used to optimise the energy consumption of a system by reducing the CPU frequency

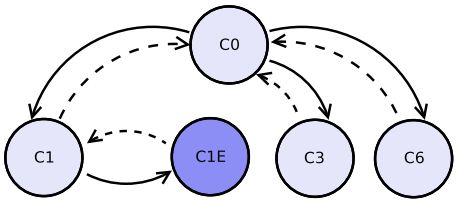


Figure 1: Possible C state transitions for the Core i7. All transitions must go through C0 except the special C1E state, which is used when all cores enter C1.

when it is determined that a lower-performing system would be acceptable. Unfortunately, static power is unaffected by frequency and a longer run time resulting from a lower clock rate increases the energy consumed through static power. Static power draw is increasing in modern systems and is a major factor contributing to the declining effectiveness of DVFS [5].

The Core i7 processor in the Dell Vostro also has a feature called *TurboBoost*, which increases the frequency of one or more cores if sibling cores are idle. The processor is sold as a 2.93 GHz model, however, the frequency of one or more cores can be increased up to 3.6 GHz (in steps) depending on the power requirements and heat dissipation of the processor. This can improve single-threaded workload performance. When and how much the frequency is increased is managed by firmware.

3.2 Idle states (C states)

Modern processors tend to offer multiple idle states often referred to as ACPI C states. These are denoted by Cx where x is a number from 0 to some maximum. Higher x values yield a *deeper* idle state, resulting in lower power consumption, but higher entry and exit latencies. C states are defined at the thread, core and package level, with $C0$ being the state in which a core is executing instructions. Beyond that there is no specification of C states, allowing manufacturers freedom to choose implementation techniques.

Because a single processor can have multiple cores and multiple hardware thread contexts (HyperThreads), constraints exist between thread, core and package idle states. There are also constraints on state transitions, as shown in Figure 1 for the Core i7 processor.

For the Core i7 and Atom Z550, Intel defines several C states which progressively apply more power-saving techniques in order to reduce power draw. For example, on the Core i7, entering C1 simply uses clock-gating to reduce processor activity, while entering C3 causes a core's local L2 cache to be flushed to the shared L3 cache and then powered down. When entering C6, a core's power supply is completely shut off, reducing leakage as well. To illustrate the impact of C state usage on system power draw, Figure 2 shows the *total* system power draw for the Dell Vostro 430s when all cores in the Core

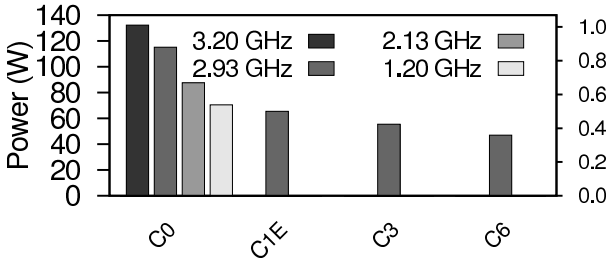


Figure 2: Idle power draw for the Dell Vostro 430s when all four cores of its Core i7 processor are placed in the same C state. Normalised scale on right-hand side.

i7 are placed in the same C state. C0 is essentially a tight idle loop, thus the power drawn in C0 is frequency dependent.

In contrast, ARM does not specify any C states for the OMAP 4430—it is left to the OS designer to choose (based on hardware constraints) what parts of the processor to power down and to what level.

Overhead from C state usage varies for a number of reasons. Firstly, in deeper C states, more power-reducing actions are taken, such as cache flushes (allowing caches to be powered down) and voltage/frequency changes. Flushing caches takes time as cache lines are written to backing stores, and voltage/frequency changes take time as voltage regulators settle and PLLs relock. Secondly, workload characteristics determine C state transition rates. A higher rate of transitions will cause higher overhead. Thirdly, the workload's memory access pattern will determine how much overhead results from cache flushes. Workloads which already have a high cache-miss rate will not be affected as much as workloads with a low miss-rate that still rely on the reuse of cached data.

4 Experimental Methodology

As discussed in Section 1, the SPEC CPU workloads that are commonly used for energy efficiency studies are not representative of the workloads that are run on most real systems. Real systems usually exhibit some level of idleness, allowing CPU sleep states to be used frequently. In contrast, the SPEC CPU workloads are CPU intensive, never allowing the CPU to idle during execution.

Therefore, we chose three real-world workloads that we believe cover a range of workload scenarios:

- MPEG playback using *mplayer* and *gstreamer*,
- serving web-pages using *Apache*, and
- the SPEC JBB2005 Java benchmark.

The first is a common workload for mobile and desktop systems and, being a single-threaded soft real-time task, it creates prolonged periods of idleness. The second

is a common multi-threaded server workload, which creates idleness due to the sporadic nature of web requests. Finally, the third is another multi-threaded server workload, but is written using Java and runs inside a Java virtual machine (JVM). These three workloads all allow the CPU's cores to enter idle states during their execution, thus allowing the effectiveness of both DVFS and different C states to be examined. Furthermore, these tasks all run for a fixed length of time regardless of CPU frequency. This means we can analyse the energy efficiency of each of them running at different CPU frequencies without having to account for static energy consumption (by padding) due to different execution times.

The three test systems (as shown in Table 1) are connected to their power sources through a power meter. As a result, all energy consumption data we report is for the total system.

We ran Ubuntu Linux (10.10) on each system, with kernel version 2.6.35 and use the *cpuidle* framework to measure C state transition rate. We used a custom cpuidle governor to allow us to choose the C state that would be used. We ran ten iterations of each benchmark, and averaged these results to obtain the final result. Standard deviation was less than 1% of the mean.

For the MPEG decode/playback workload, we decoded and played the first 60 seconds of an H.264 video on each of the platforms. We used a high-definition (HD) movie on the Core i7, and a lower resolution movie on the two embedded platforms, since due their reduced computational performance, they were unable to decode the HD movie on the applications processor without dropping frames and losing sync.

For the web-server workload, we used the ERTOS public website (http://www.ertos.nicta.com.au) as the web-server root, which contains a mix of static HTML pages and large PDF files. Tests were run over 10 minute intervals to allow for a warm-up period, allowing data to be brought from disk to the buffer-cache in memory. We used *Siege* [3] on two separate machines to generate load on the test system.

For the SPEC JBB2005 workload, we used from 1–4 warehouses. Throughput decreased as the number of warehouses was increased past four. Tests ranged from 0.5–4 minutes depending on the number of warehouses.

5 Results

In the following sections, we analyse the results from each of the three workload types. Graphs for some workloads and platforms are omitted for brevity.

5.1 MPEG playback workload

On the Core i7, decoding a high-definition (HD) MPEG stream resulted in very low CPU utilisation of between 7–17 % depending on CPU frequency. Due to the single-threaded nature of the MPEG decoding workload, only a single core could be utilised and all other cores could reside in a deep C state. The C state transition rate was approximately one transition every 10 ms, which is an order of magnitude lower than the web-server workload. As a result, negligible overhead was observed when deep C states were used, as shown in the top graph of Figure 3. There was a small anomaly in system load when using TurboBoost, as due to higher power draw in C0, there were fewer opportunities for TurboBoost to be invoked, resulting in slightly higher system load when C0 was used.

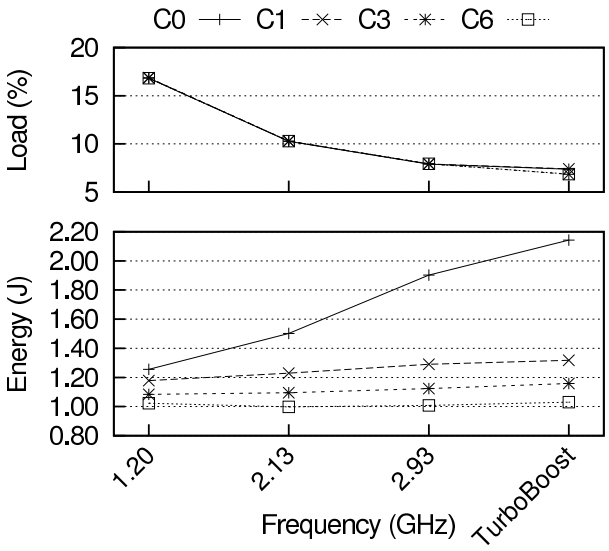


Figure 3: System load (top) and normalised energy consumption (bottom) when playing an HD-quality movie on the Vostro Desktop (Core i7) at several CPU frequencies with different C states.

As shown in the bottom graph of Figure 3, when the C0 C state was used, reducing the CPU frequency using DVFS on this platform resulted in significant energy savings. However, as deeper C states were used, DVFS became much less effective at reducing energy consumption. All data was normalised to the lowest observed energy consumption at 2.13 GHz using C6. No noticeable reduction in playback quality (dropped frames or loss of audio/video sync) was observed.

MPEG decoding and playback on the two embedded platforms was more interesting, as their processors have dedicated MPEG decode acceleration hardware (i.e. a DSP). Figure 4 shows energy consumption when several different power-management scenarios were used, including the Linux *ondemand* and *conservative* governors. Data is normalised to the maximum CPU frequency of each platform (circle points). As the bottom graph shows, using the DSP (as well as reduced CPU frequency) on the OMAP to decode the MPEG stream

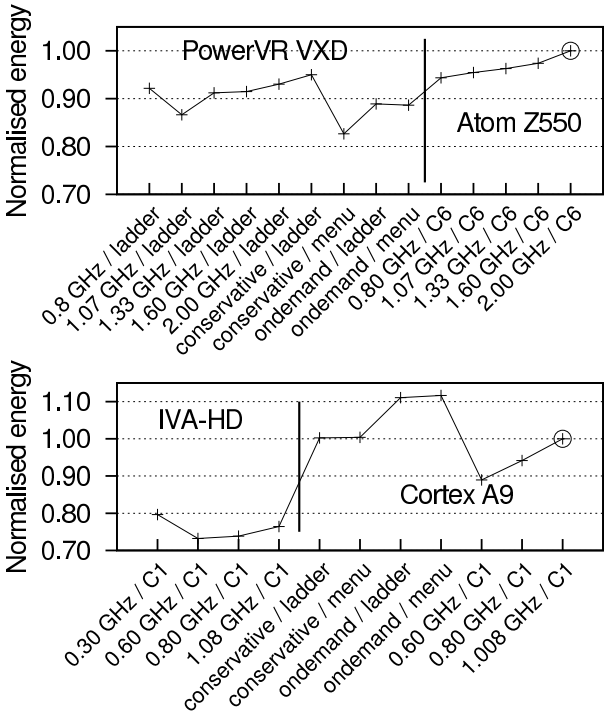


Figure 4: Normalised energy consumption of the fitPC (Atom, top) and the Pandaboard (OMAP, bottom). Points in the left-hand side are using the DSP to decode, points in the right-hand side are using the CPU.

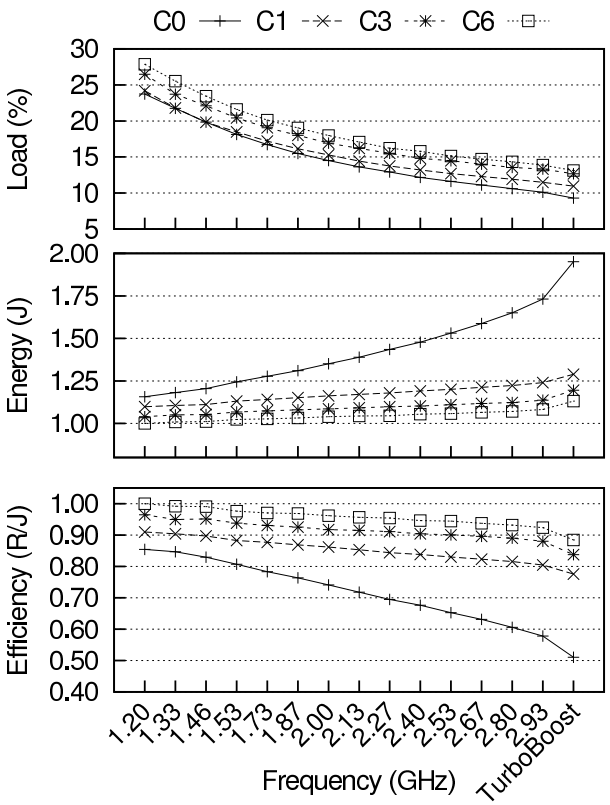


Figure 5: System load (top), normalised energy consumption (middle) and energy efficiency in requests per Joule (bottom) for Apache on the Dell Vostro (Core i7).

resulted in energy savings of up to 25%, with no loss in playback quality. In fact, these DSPs are designed to decode HD-quality MPEG streams, which the CPU's on these platforms are incapable of doing in real-time—at 0.3 GHz on the OMAP, mplayer was forced to drop frames to maintain sync, thus we omit that point. Using the DSP on the Atom also resulted in significant energy savings of up to 18 % when combined with *conservative* and *menu* on the fitPC2.

5.2 Apache web-server workload

As Barroso and Hölzle found, Google's servers spend most of their time under-utilised [4]. Therefore, to test the effectiveness and impact of DVFS and C state usage on this workload, we used Siege to generate a request rate that resulted in an under-utilised system. The top graph in Figure 5 shows that CPU utilisation on the Dell Vostro was between 12–28% depending on the CPU frequency and C state used. We observed a higher rate of C state transitions—greater than once per millisecond, more than ten times as frequent as was observed with the MPEG playback workload. As a result, using the C3 and C6 C states caused slightly higher system load. Despite this, total system energy consumption was minimised at the lowest CPU frequency (1.20 GHz) using the deepest C state (C6) as shown in the middle graph. Additionally, we found that reducing the CPU frequency had no measured impact on either throughput or response latency. As a result, energy efficiency (in requests per Joule) was maximised at the lowest CPU frequency, using the deepest C state, as shown in the bottom graph of the figure.

Similarly to the MPEG workload, we found that when deeper C states were used, the effectiveness of DVFS at improving energy efficiency was diminished. As C states improve in the future, this will become even more marked.

Using embedded-class systems in the data-centre is becoming a hot topic. Therefore, we also ran the Apache workload on the fitPC2 and Pandaboard. We found similar results to the Vostro Desktop. However, the throughput achieved on these systems was much lower than on the Vostro, and resulted in energy efficiency being significantly lower. Given that these platforms were not designed for this purpose—the Pandaboard uses a USB network interface—it is unclear whether low-power processors will have a significant impact on energy efficiency in the data-centre. Further investigation is required.

5.3 SPEC JBB2005 workload

The SPEC JBB2005 workload is a throughput-oriented benchmark. It stresses the CPU and memory

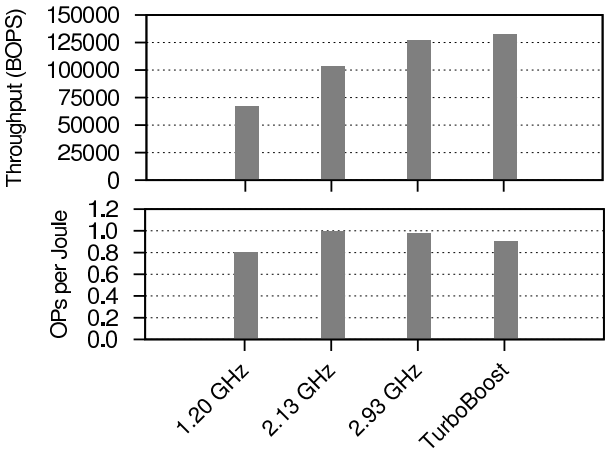


Figure 6: Throughput in billions of operations per second (top) and normalised energy efficiency in operations per Joule (bottom) of SPEC JBB2005 with four warehouses on the Vostro 430s using the C6 sleep state at several CPU frequencies.

hierarchy and is also designed to test the scalability of SMP systems. This resulted in a much higher level of CPU utilisation than the MPEG playback or web-server workloads. The total CPU utilisation was also dependent on the number of warehouses that were used. Using a single warehouse resulted in a system load of approximately 30 % spread over all cores. As the number of warehouses was increased, CPU utilisation also increased to a point where contention on other resources resulted in a bottleneck. The maximum CPU utilisation we observed with four or more warehouses on the quad-core Core i7 was about 90 %. CPU frequency had a negligible effect on the level of CPU utilisation, but throughput was impacted when CPU frequency was reduced, as shown in the top graph of Figure 6. We found that energy-efficiency (shown in the bottom graph) was maximised at 2.13 GHz using the C6 C state and four warehouses. C state transition frequency was similar to the MPEG workload, and, as a result, negligible overhead was observed when C6 was used.

6 Conclusions

We have extended previous work on energy-efficiency optimisation with DVFS by looking at realistic workloads on different classes of systems based on recent processors. Using these workloads rather than SPEC CPU benchmarks, we have shown that DVFS can still improve energy efficiency on systems that are under-utilised. This suggests that simple approaches to DVFS based on system load, like those taken by the Linux *ondemand* governor, should perform well. We also found that use of deep C states had only a small negative effect on performance, but significantly improved energy efficiency. The important factors to consider are: is the system is under-utilised and, will scaling the CPU frequency affect throughput or latency (QoS). For the MPEG playback and web-server workloads, we found that reducing CPU frequency and using the deep C states had no measurable impact on either, resulting in improved energy efficiency. This suggests that DVFS could be beneficial in the datacenter where servers must be provisioned based on the expected worst-case load, and therefore spend most of the time under-utilised. Further investigation is needed to confirm this finding for different web-server workloads, such as those that are highly dynamic and database driven. This is left as future work. We found that the SPEC JBB workload was different because CPU utilisation was independent of CPU frequency. This resulted in significantly lower throughput when CPU frequency was reduced.

From our analysis, it appears that system-level energy efficiency can be improved by both slowing the CPU down *and* using deep sleep states. However, the trends we previously identified [5] will continue, and, as a result, we will surely have to come back to this question in the future.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] Intel processor specifications, retrieved December 2010. http://ark.intel.com/.

[2] OMAP 4430 technical reference manual, Texas Instruments. http://focus.ti.com/lit/ml/swpt034a/swpt034a.pdf.

[3] Siege, an HTTP load testing and benchmarking utility. http://www.joedog.org/index/siege-home.

[4] Barroso, L. A., and Hölzle, U. The case for energy-proportional computing. *IEEE Comp. 40*, 12 (Dec 2007), 33–37.

[5] Le Sueur, E., and Heiser, G. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *2010 HotPower (HotPower'10)* (Vancouver, Canada, Oct 2010).

[6] Merkel, A., and Bellosa, F. Resource-conscious scheduling for energy efficiency on multicore processors. In *5th EuroSys Conf.* (Paris, France, Apr 2010).

[7] Miyoshi, A., Lefurgy, C., Hensbergen, E. V., Rajamony, R., and Rajkumar, R. Critical power slope: understanding the runtime effects of frequency scaling. In *16th Int. Conf. Supercomp.* (New York, NY, USA, Jun 2002), ACM Press, pp. 35–44.

[8] Snowdon, D. C., Le Sueur, E., Petters, S. M., and Heiser, G. Koala: A platform for OS-level power management. In *4th EuroSys Conf.* (Nuremberg, Germany, Apr 2009).

[9] Weiser, M., Welch, B., Demers, A. J., and Shenker, S. Scheduling for reduced CPU energy. In *1st OSDI* (Monterey, CA, USA, Nov 1994), pp. 13–23.

[10] Weissel, A., and Bellosa, F. Process cruise control—event-driven clock scaling for dynamic power management. In *CASES* (Grenoble, France, Oct 8–11 2002).

Low Cost Working Set Size Tracking *

Weiming Zhao[1], Xinxin Jin[2], Zhenlin Wang[1], Xiaolin Wang[2], Yingwei Luo[2], and Xiaoming Li[2]

[1]Dept. of Computer Science, Michigan Technological University
[2]Dept. of Computer Science and Technology, Peking University

Abstract

Efficient memory resource management requires knowledge of the memory demands of applications or systems at runtime. A widely proposed approach is to construct an LRU-based miss ratio curve (MRC), which provides not only the current working set size (WSS) but also the relationship between performance and target memory allocation size. Unfortunately, the cost of LRU MRC monitoring is nontrivial. Although optimized with AVL-tree based LRU structure and dynamic hot set sizing, the overhead is still as high as 16% on average. Based on a key insight that for most programs the WSSs are stable most of the time, we design an intermittent tracking scheme, which can temporarily turn off memory tracking when memory demands are predicted to be stable. With the assistance of hardware performance counters, memory tracking can be turned on again if a significant change in memory demands is expected. Experimental results show that, by using this intermittent tracking design, memory tracking can be turned off for 82% of the execution time while the accuracy loss is no more than 4%. More importantly, this design is orthogonal to existing optimizing techniques, such as AVL-tree based LRU structure and dynamic hot set sizing. By combining the three approaches, the mean overhead is lowered to only 2%. We show that when applied to memory balancing for virtual machines, our scheme brings a speedup of 1.85.

1 Introduction

Modeling the relationship between physical memory allocation and performance is indispensable for optimizing memory resource management. As early as the 1970s, working sets were considered as effective tools for modeling memory demands [1]. Because working sets provide a desirable metric for memory management, many solutions have been proposed to track them. One widely proposed approach is to build page-level LRU-based miss ratio curves (MRCs). This approach tracks memory accesses, and constructs a miss ratio curve to correlate memory allocation with page misses. Studies show that this approach can estimate not only the current working set size (WSS) but also the performance impact when the system or application's memory allocation is varied [2, 3, 4, 5].

However, the runtime overhead of maintaining such miss ratio curves is nontrivial. Especially because the complexity and data size of modern applications increase dramatically, the overhead of MRC tracking may overshadow its potential benefits. For example, for `SPEC CPU2006`, using a simple linked-list-based implementation, the overall execution time is increased by a factor of 1.73. Although some previous research optimizes MRC monitoring in terms of data structures [4], the overhead is still considerably high.

This paper introduces a low cost working set size tracking approach. We have implemented an AVL-based LRU structure and dynamic hot set sizing (DHS), which are detailed in our technical report [6], to lower the tracking overhead. However, our experiments show that it is still as high as 16% on average.

By taking advantage of the phase behavior of programs, we further design a novel technique, *intermittent memory tracking* (IMT), to lower the overhead without a significant loss of accuracy. This idea is based on the fact that the execution of a program can be divided into *phases*, within each of which, the memory demands are relatively stable [1]. Thus, when the monitored system or process is predicted to stay in a phase, the memory tracking can be temporarily disabled to avoid tracking cost. Later on, when a phase change is predicted to occur, the memory tracking is resumed to track the working

*Supported by NSF Career CCF0643664, the 973 Program of China No. 2007CB310900, NSFC No. 90718028 and No. 60873052, the 863 Program No.2008AA01Z112, and MOE-Intel Information Technology Foundation under No. MOE-INTEL-10-06. Many thanks to Carl Waldspurger for shepherding this paper.

set size of the new phase.

The key challenge is to predict phase changes when memory tracking is off. Fortunately, we observe that the stability of memory demands is closely correlated with that of some hardware events such as data TLB misses, and L1 and L2 cache misses. This inspired us to utilize these hardware events to predict phase changes when memory tracking is off. However, DTLB and cache-level events show much higher fluctuations (noise) than memory demands, which challenge the accuracy of phase prediction. Worse yet is that the noise level varies by application or even different phases of the same program.

To solve this problem, we design a quick self-adaptive mechanism which can adaptively select a phase-detection threshold. Experimental results show that, during an average of 82% of the time of program execution, memory tracking can be turned off and mean relative error is merely 3.9%.

2 Background and Related Work

2.1 Working Set and Miss Ratio Curve

The active working set of an application refers to the set of pages that it has referenced during the recent working set window. Knowing the working set size (WSS) enables memory resources to be utilized more efficiently. Many approaches have been proposed to estimate the WSS. VMware ESX server adopts a sampling strategy [7]. During a sampling interval, accesses to a set of random pages are monitored. By the end of sampling period, the page utilization of the set is used as an approximation of global memory utilization. This technique can tell how much memory is inactive but it cannot predict the performance impact when memory resources are reclaimed. Geiger [5] detects memory pressure and calculates the amount of extra memory needed by monitoring disk I/O and inferring major page faults. However, when the memory is over-allocated, it is unable to tell how to shrink the memory allocation.

In addition to the current WSS of a system, when memory resource competition occurs, in order to achieve optimal overall performance, we also need to know how performance would be affected by varying the memory allocation size. The miss ratio curve (MRC) that plots the page miss ratio against various amounts of available memory allocation solves the problem. Given an MRC, we can redefine WSS as the size of memory that results in less than a predefined tolerable page miss rate.

A common method to calculate an MRC is the stack algorithm [2]. The stack orders the page numbers based on their recency of accesses. Each stack entry *i* is associated with a counter, denoted as *Hist(i)*. When a reference hits a page, its stack distance, *dist*, is computed, then *Hist(dist)* is incremented by one, and finally this page is moved to the top of the stack. From *Hist*, the page miss ratio with respect to various memory allocation sizes can be computed. Constructing an MRC requires capturing or sampling a sufficient amount of page accesses. Previous research traced MRCs through a permission protection mechanism in the OS or hypervisor [8, 3, 4]. The OS or hypervisor can revoke access permission of pages, so the next accesses to those pages will cause page faults and be captured to build the MRC. For each page interception, the overhead mainly comes from page fault handling and the operation to find the stack distance which is bounded by the WSS. Zhou *et al.* [3] also proposed a hardware-based approach, but it needs extra circuits.

Hypervisor exclusive cache [9] uses an LRU-based MRC to estimate the WSS for each virtual machine. The overhead of MRC construction is analyzed but not quantified in this work. MEB [8] also uses the permission protection mechanism to build the WSS for each VM to support memory balancing. However, the overhead from MRC monitoring is significantly high, especially for applications with poor locality and very large WSSs. For example, `Gems.FDTD` in `SPEC CPU2006` exhibits a 238% overhead. To optimize cache utilization, Zhang *et al.* [10] propose to identify hot pages through scanning the page table of each process using "locality jumping" as an optimization. However, the cost of monitoring a virtualized OS is not evaluated.

2.2 Phase Prediction

Most programs show a typical phasing behavior where the program behavior in terms of IPC, branch prediction, memory access patterns, etc. is stable within a phase while there exists disruptive transition between phases [1, 11].

Shen *et al.* [11] predict locality phases by a combination of offline reuse distance profiling and runtime signal processing. Sherwood *et al.* [12] identify different phases by profiling basic block frequency and using Fourier analysis to filter out noise. However, for online phase detection, the methods that require profiling and sophisticated signal processing techniques are inappropriate. RapidMRC [13] estimates the L2 cache MRC by utilizing the PowerPC-specific Sampled Data Address Register. It selects L2 cache miss rate as the parameter to detect a phase change.

3 Intermittent Memory Tracking

Most programs show typical phasing behavior in terms of memory demands. Within a phase, the WSS remains nearly constant. This inspired us to temporarily disable memory tracking when the monitored program enters a

stable phase and re-enable it when a new phase is encountered. Through this approach, the overhead can be substantially lowered. However, when memory tracking is off, the memory tracking mechanism itself is unable to detect phase transitions anymore. Hence, an alternative method is required to wake up memory tracking when it predicts a phase change.

We find that a phase change of WSS tends be accompanied by sudden changes of the occurrences of memory-related hardware events like TLB misses, L2 misses, etc. And when the WSS remains stable, the activeness of those events is relatively stable too. These events can be monitored by special registers (Performance Monitor Counters, PMCs) built into most modern processors and accessed with negligible overhead. The key challenge is to differentiate phase changes from random fluctuations.

We propose a simple yet effective algorithm to detect behavior changes for both memory demands and performance counters. First, a moving average filter is applied for signal de-noising. Let v_i denote the sampled value (WSS or the number of occurrences of some hardware event) during ith time interval. We pick $f(i) = (v_i + v_{i-1} + \ldots + v_{i-k+1})/k$ as the filtering function to smooth the sampled values, in which k is the filtering parameter, an empirical value. When the moving average filter has not been filled up with k data, memory tracking is always enabled. Once enough data have been sampled, let v_j be the current sampled value and let $f_{mean} = mean(\{f(x)|x \in (j-k, j]\})$, $err_r = f(j)/f_{mean}$ and $err_a = |f(j) - f_{mean}|$. err_r is the relative difference between the current sampled value (smoothed) and the average of history data in the window and err_a is the absolute difference between the two. If $err_r \in [1-\mathbf{T}, 1+\mathbf{T}]$, where $\mathbf{T}$ a small threshold of choice discussed later, we assume the input signal is in a stable phase. Otherwise, we assume that a new phase is encountered. In this case, all the data in the moving average filter is cleared so the data that belong to the previous phase will not be used.

Fixed-Threshold Phase Detection $\mathbf{T}$ is the key parameter in phase detection. We first propose a scheme that uses a fixed $\mathbf{T}$. One phase detector, based on past WSS, checks if the memory demands reach a stable state so the WSS tracking can be turned off. The other detector uses PMC values to check if a new phase is seen so the WSS tracking should be woken up.

For the stability test of WSS, $\mathbf{T}$ can be set to a small value (0.05 in our evaluation) to avoid accuracy loss. In addition, err_a can also be used to guide memory tracking. For example, if memory tracking is at a MB granularity, then as long as $err_a < 1$MB, WSS can still be assumed in a stable state even when $err_r > \mathbf{T}$.

For phase detection of hardware performance events, an over-strict threshold may cause memory tracking to be enabled unnecessarily and thus undermine performance. On the other hand, if the threshold were too large, WSS changes would not be detected, causing inaccurate tracking results. Our experiments show that, for a given hardware event, the appropriate $\mathbf{T}$ may vary between programs or even vary between phases for the same program. In practice, an empirical value of $\mathbf{T}$ can be used though it may not be the optimal one.

Adaptive-Threshold Phase Detection To improve upon fixed-threshold phase detection, we propose a self-adaptive scheme which adjusts $\mathbf{T}$ dynamically to achieve better performance. The key is to feed the current stability of WSS back to the hardware performance phase detector to construct a closed-loop control system, as illustrated in Figure 1. Initially, the PMC-based phase detector can use the same threshold as used in fixed-threshold phase detection. When memory tracking is on, its current stability is computed and compared with the PMC-based phase detector's decision.

If both of the results are consistent, nothing will be changed. If the current memory demands are stable, while the PMC-based detector makes the opposite decision ($err_r > \mathbf{T}$), it implies that the current threshold is too tight. As a result, its $\mathbf{T}$ is relaxed to its current err_r. Next time, with increased $\mathbf{T}$, the PMC-based detector will most likely find that the system enters a "stable" state and thus turn off memory tracking. On the contrary, if the current memory demands are unstable, while the PMC-based phase detector assumes stable PMC values, i.e. $err_r < \mathbf{T}$, it implies an over-relaxed threshold. Thus, its current $\mathbf{T}$ is lowered to err_r. In short, when the WSS is stable and memory tracking is on, it is only because the PMC-based phase detector is overly sensitive. As a result, $\mathbf{T}$ will be increased until PMC values are considered to be stable too. Then, memory tracking will be turned off.

However, when memory tracking is off, this self-calibration is paused as well, which might miss the chance to tighten the threshold as it should had memory tracking been on. To solve this problem, we introduce a checkpoint design. When memory tracking has been disabled for $ckpt$ consecutive sampling intervals, it is woken up to check if $\mathbf{T}$ should be adjusted or not. If no adjustment is needed, it will be turned off again until it reaches the next checkpoint or meets a new phase. The value of $ckpt$ is adaptive. Initially, it is set to some pre-defined value $ckpt_{init}$. Afterward, if no adjustment is made in the previous checkpoint, it can be increased by some amount ($ckpt_{step}$) until it reaches a maximum value $ckpt_{max}$. Whenever an adjustment is made, $ckpt$ is restored to $ckpt_{init}$. In the ideal case, the ratio of the time that memory tracking is on to the whole execution time, called *up ratio*, is nearly $1/ckpt_{max}$.

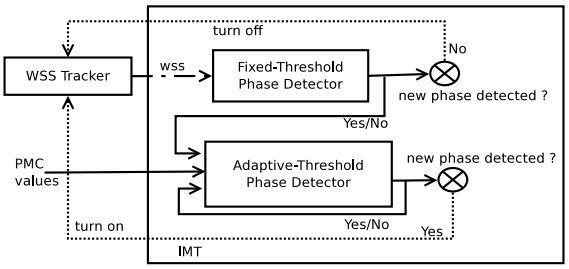


Figure 1: Adaptive-Threshold IMT

4 Implementation and Evaluation

To verify the effectiveness of our WSS tracking and evaluate its application in virtualized environments, we use the Xen 3.4 [14] hypervisor, an open source virtual machine monitor, as the base of our implementation. When a para-virtualized guest OS that runs in user mode attempts to modify its page tables, it has to make a hypercall to explicitly notify the hypervisor to do the actual update. In our modified hypervisor, once such requests are received, our code will first perform the requested update as usual and then revoke the access permission by setting the corresponding bit on the page table entry. For hardware-assisted virtualized machines (HVM), this permission revoking mechanism can be done during the emulation of page table writing or propagation of guest page tables to shadow page tables. Later on, if the guest OS attempts to access that page, it will trigger a minor page fault, which will trap into the hypervisor first. In the modified page fault handling routine, the miss ratio curve is updated for that access and permission is restored.

To verify the effects of our WSS tracking in memory balancing, we use the VM memory balancer that was implemented in [8]. Both IMT and the memory balancer run on Dom-0, a privileged virtual machine. IMT is written in about 300 lines of Python code, with a small C program to initiate hypercalls. Via customized hypercalls, IMT communicates with the WSS tracker to receive current WSS estimation and PMC counter values and send its decisions to the WSS tracker. Based on the assumption that the memory access pattern is nearly unchanged in a stable phase, when the WSS tracker is woken up, it uses the same LRU list and histogram as in the last tracking interval. In our experiments, WSS and PMCs are sampled every 3 seconds. For checkpointing, its initial value ($ckpt_{init}$), the increment, and $ckpt_{max}$ are set to 10, 5, and 20 sampling intervals, respectively, which means the minimum up ratio is nearly 0.05.

All experiments are performed on a server equipped with one 2.8 GHz Intel Core i5 processor (4 cores with HT enabled) and 8 GB of 800 MHz DDR2 memory. Each virtual machine runs 64-bit Linux 2.6.18, configured with 1 virtual CPU and 3 GB of memory (except in the memory balancing test). We select a variety of benchmark suites, including the `SPEC CPU2006` benchmark suite and `DaCapo` [15], a Java benchmark suite, to represent real world workload and evaluate the effectiveness of our work.

$\mathbf{T} = 0.05$		$\mathbf{T} = 0.2$		$\mathbf{T} = 0.3$		Adaptive	
UR	MRE	UR	MRE	UR	MRE	UR	MRE
.27	.057	.13	.100	.11	.126	.11	.039

Table 1: Mean Up Ratios and MREs of SEPC 2006

In this section, we first evaluate the performance of IMT with various configurations. However, even for the same program, its WSS may not be identical among all runs, which undermines the fairness of comparison. Besides, if IMT is actually used and when it turns off memory tracking, the accuracy loss caused by IMT cannot be precisely measured. As a result, we run IMT against simulated inputs. Then we examine the overhead of WSS tracking with actual runs. Finally, we design a scenario to demonstrate the application of WSS tracking.

4.1 Performance of IMT

The performance of IMT is evaluated by two metrics: (1) the time it saves by turning off memory tracking, reflected by *up ratio*, and (2) the accuracy loss due to temporary inactivation of memory tracking, indicated by *mean relative error*. We first run each benchmarks and sample the WSS and PMC values every 3 seconds without IMT. Then, we feed the trace results to the IMT algorithm to simulate its operations. That is, given inputs $\{M_0, \ldots, M_i\}$ and $\{P_0, \ldots, P_i\}$, the IMT algorithm outputs m_i, in which M_i and P_i are the i-th memory demand and i-th PMC value sampled in the trace results, respectively, and m_i is the estimated memory demand. When the IMT algorithm indicates the activation of memory tracking, $m_i = M_i$, otherwise, $m_i = M_j$ where j is the last time that memory tracking is on. Given a trace with n samples, its mean relative error is computed as $MRE = (\sum_{i=1}^{n} \frac{|M_i - m_i|}{M_i})/n$.

To evaluate the performance of fixed and adaptive thresholds for IMT, we use a DTLB miss as the hardware performance event for phase detection. We have indeed examined three memory related hardware events, DTLB misses, L1 references, and L2 misses as well as their combinations, for phase detection. Interestingly, there is no obvious difference both in accuracy and up ratio [6]. For fixed thresholds, $\mathbf{T}$ varies from 0.05 to 0.3, two extreme ends of the spectrum. Table 1 shows mean up ratios and MREs of `SPEC CPU2006`. The results of individual programs are presented in [6].

Using fixed thresholds, when $\mathbf{T} = 0.05$, memory

tracking is off nearly three fourths of the time with an MRE of about 6%. When **T** is increased to 0.3, memory tracking is activated for only about one tenth of the time, while the MRE increases to 13%. With adaptive thresholds, its up ratio is nearly the same as that of $\mathbf{T} = 0.3$, while its MRE is even smaller than that of $\mathbf{T} = 0.05$. Clearly, adaptive thresholding outperforms the fixed-threshold algorithm.

Figure 2 shows the results of several cases using adaptive-threshold IMT. The upper parts of each figure show the status of memory tracking: a high level means it is enabled and a low level means it is disabled. In the bottom parts, thick lines and thin lines plot the WSS and normalized data TLB misses from the traces (sampled without IMT), respectively. Dotted lines plot the WSS assuming IMT is enabled. Figure 2(a) shows the common case where there are multiple phases in terms of WSS and DTLB misses. In Figure 2(b) and Figure 2(c), two representative cases, where checkpointing and adaptive thresholding take effect, are presented. For Figure 2(b), when examined from an overall scope, the WSS varies gradually. However, the WSS looks more stable when examined from each small time window. This makes the program assume that the WSS is in the stable mode and thus turns off memory tracking. Nonetheless, with the checkpointing mechanism, the WSS variances are still captured. Figure 2(c) shows that, though the WSS is stable most of the time, the DTLB miss fluctuates randomly. With the adaptive algorithm, the noise is filtered by increased thresholds.

With adaptive-threshold IMT, `429.mcf` shows an MRE of 38.7% while all others are less than 8% with a mean of 2%. For `429.mcf`, as Figure 2(a) shows, most of the time, the WSS estimation using IMT follows the one without using IMT. The high relative error is because its WSS changes dramatically up to 9 times at the borders of phase transitions. Though after a short delay, IMT detects the phase change and wakes up memory tracking, those exceptionally high relative errors lead to a large MRE. More specifically, during 67% of its execution time, the relative errors are below 4%, and during 84% of the time, the relative errors remain within 10%.

4.2 Overhead Evaluation

To evaluate the actual effects of using IMT, we measure the WSS tracking overhead on actual runs. As Table 2 shows[1], even optimized with AVL-based LRU and dynamic hot set sizing, the mean overhead of `SPEC CPU2006` is 16% due to large WSSs and/or bad locality of some programs. For example, for high-overhead programs, such as `429.mcf` and `433.milc`, the average WSSs are 859 MB and 334 MB, respectively, while the

[1]The complete list is in [6].

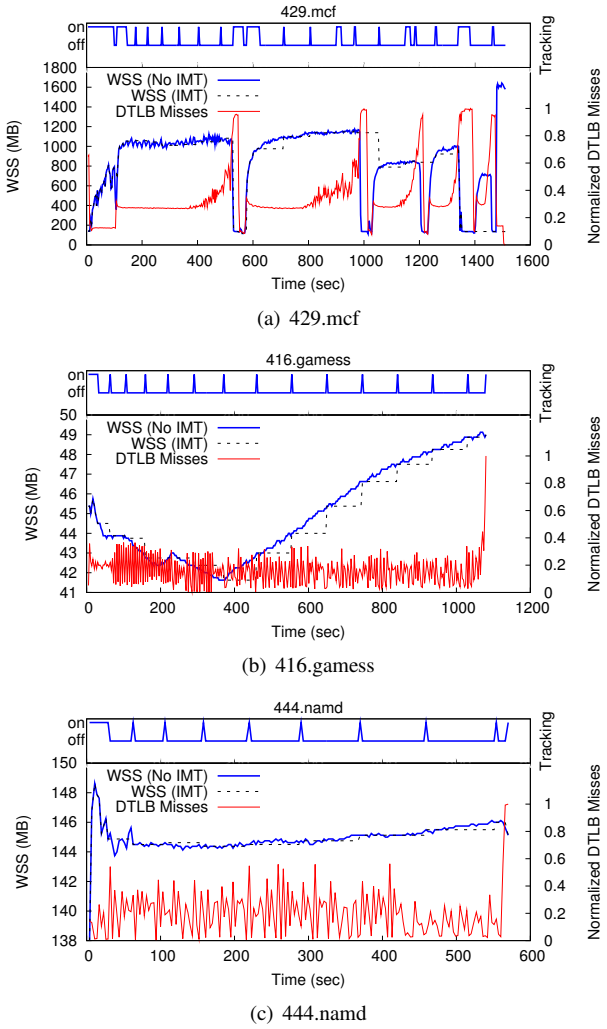


Figure 2: Examples of Using Adaptive-Threshold IMT

average WSSs of `401.bzip2` and `416.gamess` are only 24 MB and 45 MB, respectively.

Enhanced with fixed-threshold IMT ($\mathbf{T} = 0.2$), the mean overhead is lowered to 6%. Using adaptive-threshold IMT, the mean overhead is further reduced to 2% by cutting off half of the up time of memory tracking.

4.3 Applications to VM Memory Balancing

One typical scenario for WSS tracking is memory balancing. Two VMs are monitored on a Xen-based host. One VM runs `470.lbm`, meanwhile, the other VM runs `433.milc`. Initially, each VM is allocated 700 MB of memory. In the baseline setting, no memory balancing or WSS tracking is used. With memory balancing, three variations are compared: memory tracking without IMT, using IMT with a fixed threshold of 0.2 and using IMT with an adaptive threshold. Figure 3 shows the normalized speedups with memory balancing against the baseline setting. Note that, the balancer is designed to reclaim

Program	Norm. Exec. Time				Up Ratio	
	L	A+D	A+D + I_f	A+D + I_a	I_f	I_a
401.bzip2	1.03	1.02	1.01	1.01	0.76	0.14
416.gamess	1.01	1.01	1.00	1.00	0.18	0.09
429.mcf	59.16	1.75	1.41	1.04	0.72	0.37
433.milc	13.08	3.83	2.46	1.05	0.52	0.11
470.lbm	4.31	1.77	1.01	1.00	0.17	0.10
...		...			...	
Mean	2.73	1.16	1.06	1.02	0.26	0.12

Table 2: Normalized Execution Time and Up Ratios
L: linked list, A+D: ABL and dynamic hot set, I_f: fixed-threshold IMT ($\mathbf{T} = 0.2$), I_a: adaptive-threshold IMT

unused memory, so the total allocated memory to the two VMs may be less than 1400 MB.

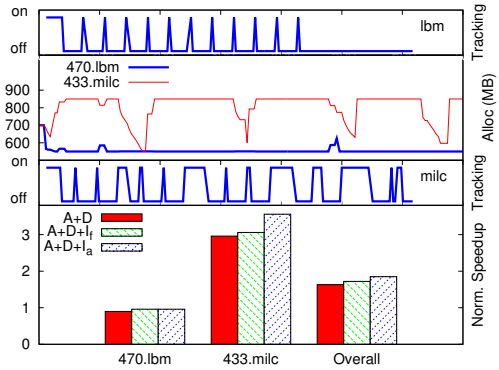


Figure 3: Speed-Ups With Memory Balancing

When balanced without IMT, the performance of `470.lbm` degrades by 10% due to the overhead of memory tracking, while the performance of `433.milc` is boosted by 2 times due to the extra memory it gets from the other VM. Using IMT, the performance impact of memory tracking on `470.lbm` is lowered to 4%. For `433.milc`, with fixed-threshold or adaptive-threshold IMT, its speedup is increased from 2.96 to 3.06 and 3.56 respectively. The overall speedups of balancing without IMT, with fixed-threshold IMT and adaptive-threshold IMT are 1.63, 1.72 and 1.85. Hence, using adaptive-threshold IMT, an additional 22% speedup is gained.

5 Conclusion and Future Work

LRU-based working set size estimation is an effective technique to support memory resource management. This paper makes this technique more applicable by significantly reducing its overhead. We present a novel intermittent memory tracking scheme. Experimental evaluation shows that our solution is capable of reducing the overhead with sufficient precision to improve memory allocation decisions. In an application scenario of balancing memory resources for virtual machines, our solution boosts the overall performance. In the future, we plan to develop theoretical models that verify the correlations among various memory events.

References

[1] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1), 1980.

[2] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[3] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS'04*, pages 177–188, 2004.

[4] T. Yang, E. D. Berger, S. F. Kaplan, and J. Eliot B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI'06*, pages 103–116, 2006.

[5] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. *SIGOPS Oper. Syst. Rev.*, 40(5):14–24, 2006.

[6] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Efficient LRU-based working set size tracking. Technical Report CS-TR-11-01, Department of Computer Science, Michigan Tech University, 2011.

[7] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[8] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE'09*, 2009.

[9] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX ATC'07*, pages 1–15, 2007.

[10] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, 2009.

[11] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS'04*, 2004.

[12] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT'01*, pages 3 –14, 2001.

[13] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating l2 miss rate curves on commodity systems for online optimizations. In *ASPLOS'09*, pages 121–132, 2009.

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, 2003.

[15] S. M. Blackburn, R. Garner, and C. Hoffman et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190, 2006.

FVD: a High-Performance Virtual Machine Image Format for Cloud

Chunqiang Tang
IBM T.J. Watson Research Center
ctang@us.ibm.com

Abstract

Fast Virtual Disk (FVD) is a new virtual machine (VM) image format and the corresponding block device driver developed for QEMU. QEMU does I/O emulation for multiple hypervisors, including KVM, Xen-HVM, and VirtualBox. FVD is a holistic solution for both Cloud and non-Cloud environments. Its feature set includes flexible configurability, storage thin provisioning without a host file system, compact image, internal snapshot, encryption, copy-on-write, copy-on-read, and adaptive prefetching. The last two features enable instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. As its name indicates, FVD is fast. Experiments show that the throughput of FVD is 249% higher than that of QCOW2 when using the PostMark benchmark to create files.

1 Introduction

Despite the existence of many popular virtual machine (VM) image formats (e.g., QEMU QCOW2 [5], VirtualBox VDI [10], VMWare VMDK [11], and Microsoft VHD [6]), FVD came out of our unsatisfied needs in the IBM Cloud [9]. FVD distinguishes itself from existing image formats in multiple aspects: *flexible configurability*, *high performance*, and *rich features*.

Flexible configurability. As virtualization becomes pervasive, virtual disks of virtual machines (VM) may be used in diverse settings. For example, the main requirement can be storage thin provisioning or high disk I/O performance. A disk image can be stored as a regular file or a logical volume. It can use direct-attached storage, network-attached storage, or storage-area network.

Existing image formats support diverse settings in a one-size-fit-all manner, by bundling all functions into one inseparable, monolithic piece. This can cause inefficiency even in common cases. Consider, for example, the copy-on-write (CoW) feature of virtual disk. QCOW2, VDI, VMDK, and VHD all mix the function of CoW dirty block tracking with the function of storage space allocation. In a common setting where the image is stored on a host file system, this leads to doing storage allocation twice (first in the image format and then in the host file system), which causes data fragmentation twice and doubles the disk I/O overhead for metadata access.

By contrast, a design principle of FVD is to make all functions orthogonal so that each function can be enabled or disabled individually, even for two virtual disks attached to the same VM. The purpose is to support diverse use cases without being burdened with the overhead of all functions. This is a significant departure from existing image formats and challenges some conventional wisdom in image format design. For the specific example above, FVD can be configured to enable CoW dirty block tracking, disable its own storage allocation, and delegate storage allocation entirely to the host OS, which has abundant options to optimize for a given workload, e.g., storing the image on a logical volume, or storing the image as a regular file in a host file system, with the choices of ext2/ext3/ext4, JFS, XFS, ReiserFS, etc.

VM mobility in a Cloud. FVD is a feature-rich, holistic solution for both Cloud and non-Cloud environments. This paper is focused on its copy-on-read and adaptive prefetching features, which improve VM disk data mobility in a Cloud.

In a Cloud like Amazon EC2, the storage space for a VM can be allocated from multiple sources, which offer different performance, reliability, and availability at different prices. In EC2, a VM is provided with 170GB or more ephemeral storage (i.e., direct-attached storage (DAS)) at no additional charge. Persistent storage (i.e., network-attached storage (NAS)) is more expensive, which is charged not only for the storage space consumed but also for every disk I/O performed. For example, if a VM's root file system is stored on persistent storage, even the VM's disk I/O on its temporary directory */tmp* incurs additional costs. As a result, it is popular to use ephemeral storage for a VM's root file system. However, using DAS slows down the process of VM creation and VM migration, which diminishes the benefits of an elastic Cloud.

The discussion below uses KVM and QEMU as examples. In a Cloud, VMs are created based on read-only image templates stored on NAS and accessible to all hosts. A VM's virtual disk can use different image formats. QEMU's RAW format is simply a byte-by-byte copy of a physical disk's content stored in a regular file. If a VM uses the RAW format, the VM creation process may take a long time and cause resource contentions, because the host needs to copy a complete image template (i.e., gigabytes of data) from NAS across a heavily shared network in order to create a new RAW image on DAS. This problem is illustrated in Figure 1(a).

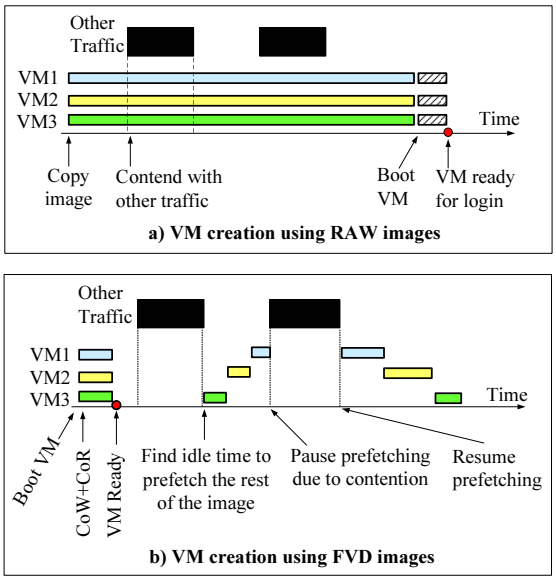


Figure 1: Comparison of the VM creation processes. This example creates three VMs concurrently.

QCOW2 [5] is another image format supported by QEMU. It does copy-on-write, i.e., the QCOW2 image only stores data modified by a VM, whereas unmodified data are always read from the base image. QCOW2 supports fast VM creation. The host can instantly create and boot an empty QCOW2 image on DAS, whose base image points to an image template stored on NAS. Using QCOW2, however, limits the scalability of a Cloud, because a large number of VMs may repeatedly read unmodified data from the base image, generating excessive network traffic and I/O load on NAS.

The solution in FVD is to do copy-on-read (CoR) and adaptive prefetching, in addition to copy-on-write (CoW). CoR avoids repeatedly reading a data block from NAS, by saving a copy of the returned data on DAS for later reuse. Adaptive prefetching uses resource idle time to copy from NAS to DAS the image data that have not been accessed by the VM. These features are illustrated in Figure 1(b).

In addition to instant VM creation, FVD also supports instant VM migration, even if the VM's image is stored on DAS. FVD can instantly migrate a VM without first transferring its disk image. As the VM runs uninterruptedly on the target host, FVD uses CoR and adaptive prefetching to gradually move the image from the source host to the target host, without user perceived downtime.

2 Overview of FVD

This paper is focused on the Cloud-inspired features of FVD, i.e., copy-on-read and adaptive prefetching. To set stage for the detailed discussion in Section 3, this section first describes how a virtual disk works today, using KVM [4], QEMU [1], and QCOW2 [5] as examples, and then presents an overview of the holistic FVD solution.

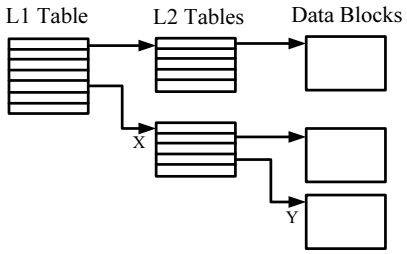


Figure 2: QCOW2's two-level lookup index.

2.1 How a Virtual Disk Works Today

When a VM issues a disk I/O request for data at a *virtual block address* (VBA), the host Linux kernel forwards the request to QEMU running in the user space. QEMU's QCOW2 driver translates the VBA into an *image block address* (IBA), which specifies where the requested data are stored in the QCOW2 image file, i.e., IBA is an offset in the image file. Specifically, QCOW2 uses the index in Figure 2 to perform the address translation. A VBA d is split into three parts, i.e., $d = (d_1, d_2, d_3)$. The d_1 entry of the L1 table points to an L2 table X. The d_2 entry of the L2 table X points to a data block Y. The requested data are located at offset d_3 of the data block Y.

Initially, a QCOW2 image contains only the L1 table, with all data blocks and L2 tables unallocated. A data block is allocated at the end of the image file upon the first write to that block. As a result, a block's IBA solely depends on when it is written for the first time, regardless of its VBA. This behavior may end up with an undesirable data layout on the physical disk. For example, when the guest OS creates a file system, it writes out the file system metadata, which are all grouped together and assigned consecutive IBAs by QCOW2, despite the fact that the metadata's VBAs are deliberately scattered for better reliability and locality, e.g., co-locating inodes and file content blocks in block groups. As a result, it may cause a long disk seek distance between accessing a file's metadata and accessing the file's content blocks. This problem is not unique to QCOW2. It exists in all popular image formats, including VDI, VMDK, and VHD.

When the guest VM reads data at the VBA d=(d_1, d_2, d_3), the QCOW2 driver determines whether the data block is allocated in the QCOW2 image by checking if the corresponding L1 or L2 table entry is empty. If so, the data are read from the base image. Otherwise the data are read from the QCOW2 image. Since the lookup index implements both dirty block tracking and storage allocation, the two functions become inseparable.

2.2 How FVD Works

Below, we summarize the on-disk metadata used by FVD to provide a diverse set of functions:

- A bitmap for implementing copy-on-write.

- A one-level lookup table for implementing storage allocation.
- A metadata journal for committing changes of the bitmap and the lookup table.
- A reference-count table for implementing internal snapshot.

Bitmap. The bitmap is enabled only if a new FVD image is created based on an existing image template (so-called *base image*). When the VM issues a disk write, the base image is not modified. Instead, the new data are saved in the FVD image. This behavior is called copy-on-write. A bit in the bitmap tracks where the latest content of a *block* is stored. The bit is 0 if the block is in the base image, and the bit is 1 if the block is in the FVD image. The unit of a block is configurable per virtual disk, with a default size of 64KB. To represent the state of a 1TB base image, FVD only needs a 2MB bitmap, which can be easily cached in memory. The bitmap also implements copy-on-read and adaptive prefetching.

Lookup table. The lookup table implements storage allocation. One entry of the look table maps a data *chunk*'s VBA to its IBA. The unit of a chunk is configurable per virtual disk, with a default size of 1MB. (Note that VDI uses 1MB chunks. VHD and the ESX version of VMDK use 2MB chunks.) For a 1TB virtual disk, the size of FVD's lookup table is only 4MB. Because of the table's small size, there is no need to use a more complicated two-level index as that in QCOW2.

Because FVD itself is capable of managing storage allocation, one valid configuration is to store an FVD image directly on a logical volume to avoid the overhead of a sophisticated host file system. This configuration still supports storage thin provisioning. The initial size of the logical volume can be small. During the execution of the VM, FVD asks the host OS to increase the size of the logical volume when more storage space is needed.

Separating the implementation of copy-on-write from the implementation of storage allocation provides several benefits. First, the lookup table can be optionally disabled to avoid the overhead and data fragmentation caused by doing storage allocation at the image level. In this case, FVD maintains a linear mapping between a chunk's VBA and IBA without any address translation, and relies on the host file system for storage allocation.

Another benefit is that it makes the metadata smaller and easier to cache, by using the bitmap to track data at the finer *block* granularity, and using the lookup table to track data at the coarser *chunk* granularity. The bitmap is small because of its efficient representation. The lookup table is small because the large chunk size leads to less table entries. For a 1TB virtual disk, FVD's bitmap and one-level lookup table together are only 6MB, whereas QCOW2's two-level lookup table is 128MB.

Metadata journal. When the bitmap and/or the lookup table need be modified, the changes are saved in the journal, as opposed to updating the bitmap and/or the lookup table directly. The journal size is configurable per virtual disk, with a default size of 16MB. When the journal is full, which happens infrequently, the entire bitmap and the entire lookup table are flushed to disk. Then the journal can be recycled for reuse. The flush avoids the overhead of fine-grained journal cleaning operations that are common in journaling file systems. The flush is quick, because the bitmap and the lookup table are small.

The journal provides several benefits. First, updating both the bitmap and the lookup table requires only a single write to the journal. Second, k concurrent updates to any potions of the bitmap or the lookup table are converted to sequential writes in the journal. Finally, it increases concurrency by allowing multiple parallel updates to the same sector in the bitmap or the lookup table.

Reference-count table. There are two ways of implementing virtual disk snapshot: external snapshot and internal snapshot. External snapshot can be easily implemented on top of any image form that already supports copy-on-write (CoW), including VMDK, QCOW2, and FVD. When the user takes a snapshot, the current image file S_{i-1} is made read-only and a new CoW image file S_i is created based on S_{i-1}. After a series of snapshots are taken, it creates a chain of dependent snapshot files $S_0 \leftarrow S_1 \leftarrow \cdots \leftarrow S_i$. Deleting a snapshot S_{j-1} in the middle of a snapshot chain can be a slow operation. Before removing the snapshot file S_{j-1}, it must physically copy from S_{j-1} to S_j those data chunks modified in S_{j-1} but not modified in S_j.

Internal snapshot avoids this problem by storing all snapshots in a single file. For each data chunk C in use, an entry in the reference-count table records the number of snapshots using C. Creating/deleting a snapshot simply amounts to incrementing/decrementing the reference count of data chunks that form the snapshot. A data chunk is free for reuse when its reference count becomes zero.

QCOW2 and FVD are the only two image formats that support internal snapshot, but they differ in implementation and performance. Conceptually, an image consists of an arbitrary number of read-only historical snapshots and a single writable current view (WCV). The WCV is the virtual disk content perceived by the running VM. QCOW2's reference-count table tracks all data chunks used by either snapshots or the WCV. Because the WCV changes as the VM runs, during normal executions of the VM, QCOW2 incurs disk I/O overhead for updating the on-disk reference-count table and memory overhead for caching the reference-count table. By contrast, FVD's reference-count table tracks chunks used by snapshots but does not track chunks used by the WCV (since the WCV is already tracked by the lookup table). Because read-only

snapshots do not change during normal executions of the VM, FVD need not update or cache the reference-count table during normal executions of the VM.

3 Using FVD's Bitmap to Support Copy-on-Write, Copy-on-Read and Prefetching

FVD is a comprehensive solution with many features. Due to the space limitation, the rest of this paper is focused on using FVD's bitmap to support copy-on-write, copy-on-read, and adaptive prefetching. The discussion below assumes that the bitmap is enabled but all other metadata (the lookup table, the metadata journal, and the reference-count table) are disabled. This configuration by itself is a functional, high-performance image format. Figure 3 shows FVD under this configuration.

3.1 Basic Read/Write Operation

With the lookup table disabled, FVD maintains a linear mapping between a block's VBA and IBA. When the VM writes to a block with VBA d, FVD stores the block at offset d of the "FVD Data Region" in Figure 3, without any address translation. FVD relies on the host OS for storage allocation. If the FVD image is stored on a host file system that supports sparse files, no storage space is allocated for a data block in the virtual disk until the VM actually writes to that block.

To start a new VM in a Cloud, the host creates an FVD image on its DAS, whose base image points to an image template on NAS. The "FVD Data Region" in Figure 3 can be larger than the base image, because an image template can be used to create VMs whose virtual disks are of different sizes, depending on how much the user pays. *resize2fs* can expand the file system in the base image to the full size of the virtual disk.

When handling a disk write request issued by the VM, the FVD driver stores the data in the FVD image and updates the bitmap to indicate that those data now are in the FVD image rather than in the base image. The bitmap-update step is skipped if the corresponding bit(s) in the bitmap are set previously. If the write request is not aligned on the block boundary, before writing the data to the image, the FVD driver reads a full block from the base image and merges it with the data to be written.

When handling a disk read request issued by the VM, the FVD driver checks the bitmap to determine if the requested data are in the FVD image. If so, the data are read from the FVD image. Otherwise, the data are read from the base image and returned to the VM. While the VM continues to process the returned data, in the background, a copy of the returned data is saved in the FVD image. Future reads for the same data will get them from the FVD image on DAS rather than from the base image on NAS. This copy-on-read behavior helps avoid generating excessive network traffic and I/O load on NAS.

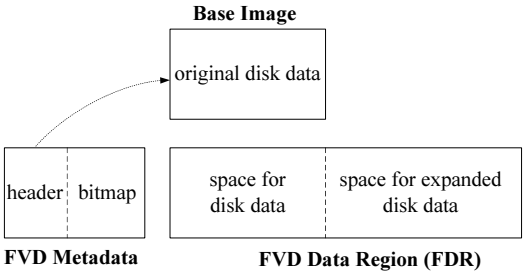


Figure 3: An simplified view of the FVD image format, with only the bitmap enabled.

3.2 Optimizations for Read/Write

Compared with the RAW image format, a copy-on-write image format always incurs additional overhead in reading and updating its on-disk metadata. Below, we summarize several optimizations that eliminate this overhead in common cases. The word "free" below means no need to update the on-disk bitmap.

In-memory bitmap: Eliminate the need to repeatedly read the bitmap from disk by always keeping a complete copy of the bitmap in memory. The bitmap is only 20KB for a 1TB FVD image based on a 10GB image template. Note that, in Figure 3, the bitmap size is proportional to the base image size rather than the FVD image size.

Free writes to beyond-base blocks: Eliminate the need to update the on-disk bitmap when the VM writes to a block residing in the "space for expanded disk data" in Figure 3. This is a common case if the base image is reduced to its minimum size by *resize2fs*. Note that 1) a minimum-sized image template has no unused free space, and 2) most data in an image template are read-only and rarely overwritten by a running VM due to the template nature of those data, e.g., program executable. Consequently, disk writes issued by a running VM mostly target blocks residing in the "space for expanded disk data" in Figure 3. Since those "beyond-base" blocks cannot reside in the base image and hence have no state bits in the bitmap, there is simply no need to update the bitmap when writing to those blocks.

Free writes to zero-filled blocks: Eliminate the need to update the on-disk bitmap when the VM writes to a block whose original content in the base image is completely filled with zeros. This is a common case if the base image is not reduced to its minimum size by *resize2fs* and has many empty spaces. This optimization is realized by using a tool to search for zero-filled blocks in the base image and preset their state bits to 1 in the FVD bitmap. This is an offline process only done once per image template.

Free copy-on-read and free prefetching: Eliminate the need to update the on-disk bitmap when the FVD driver saves a block in the FVD image due to either copy-on-read or prefetching. This does not compromise data in-

tegrity in the event of a host crash, because the block's content in the FVD image is identical to that in the base image and reading from either place gets the correct data.

Zero overhead once prefetching finishes: Entirely eliminate the need to read or update the bitmap, once all blocks in the base image are prefetched. This is because the bitmap's content is known in a priori to be 1 for all bits.

3.3 Adaptive Prefetching

FVD uses copy-on-read to bring data blocks from NAS to DAS on demand as they are accessed by the VM. Optionally, prefetching uses resource idle time to copy from NAS to DAS the rest of the image that have not been accessed by the VM. Prefetching is a resource intensive operation, as it may transfer gigabytes of data across a heavily shared network. To avoid causing a contention on any resource (including network, NAS, and DAS), FVD can be configured to limit prefetching rate and pause prefetching when a resource contention is detected.

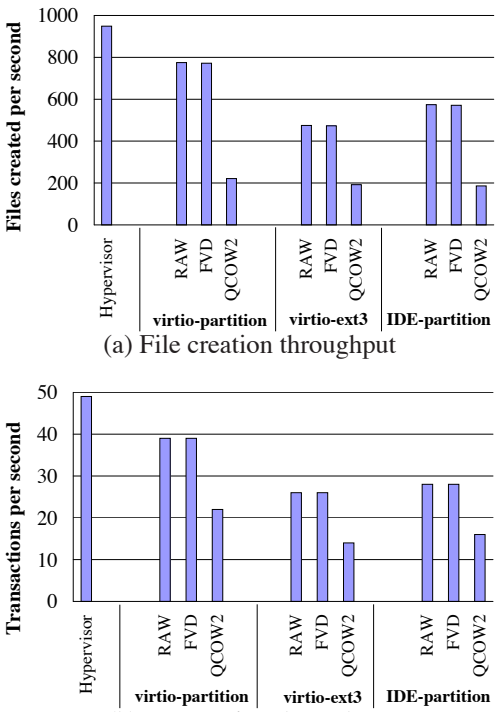
Two throughput limits (KB/s) control the behavior of prefetching data from the base image. The base image read throughput is capped at the upper limit using a leaky bucket algorithm. If the throughput drops below the lower limit, the FVD driver concludes that a resource contention has occurred. It makes a randomized decision. With a 50% probability, it temporarily pauses prefetching for a randomized period of time. If the throughput is still below the lower limit after prefetching resumes, it pauses prefetching again for a longer period of time, and so forth. Similarly, two throughput limits control the behavior of writing prefetched data to the FVD image.

4 Experimental Results

We implemented FVD in QEMU. Due to the space limitation, this paper only presents the results of one experiment. (More results are available in the longer version of this paper [8]). In this experiment, FVD's bitmap is enabled but all other metadata (the lookup table, the metadata journal, and the reference-count table) are disabled. Moreover, since QCOW2 does not support CoR and prefetching, those features are disabled in FVD in order to make a fair comparison of the basic CoW feature.

The experiment is conducted on IBM HS21 blades connected by 1Gb Ethernet. Each blade has two 2.33GHz Intel Xeon 5148 CPUs and a 2.5-inch hard drive (model MAY2073RC). The blades run QEMU 0.12.30 and Linux 2.6.32-24 with the KVM kernel modules. QEMU is configured to use direct I/O.

Figure 4 shows the performance of PostMark [3] under different configurations. The execution of PostMark consists of two phases. In the first "file-creation" phase, it generates an initial pool of files. In the second "transaction" phase, it executes a set of transactions, where each transaction consists of some file operations (creation, deletion, read, and append). In this experiment, the total size of files created in the first phase is about 50GB, and the size of an individual file ranges from 10KB to 50KB.



(a) File creation throughput

(b) Transaction throughput

Figure 4: Performance of PostMark.

In Figure 4, the "Hypervisor" bar means running PostMark in a native Linux without virtualization. The "RAW", "FVD", and "QCOW2" bars mean running PostMark in a VM whose image uses the different formats, respectively. Like that in a Cloud, a QCOW2 or FVD image V is stored on the local disk of a blade X, whereas the base image of V is stored on another blade Y accessible through NFS. The base image contains Ubuntu 9.04, and is reduced to its minimum size (501MB) by *resize2fs*. A RAW image is always stored on the local disk of a blade. The VM's virtual disk is divided into two partitions. The first partition of 1GB stores the root file system. The second partition of 50GB disk is formatted into an ext3 file system, on which PostMark runs.

For the "IDE-partition" group in the figure, the VM's block device uses the IDE interface and the VM image is stored on a raw partition in the host. For the "virtio-ext3" group, the VM's block device uses the paravirtualized *virtio* interface and the VM image is stored on a host ext3 file system, which is reformatted before each run of the experiment. For the "virtio-partition" group, it uses *virtio* and the VM image is stored on a raw partition in the host.

Figure 4 shows significant advantages of FVD over QCOW2. In the file creation phase, the throughput of FVD is 249% higher than that of QCOW2 (by com-

paring the "FVD" bar and the "QCOW2" bar in the "virtio-partition" group of Figure 4(a)). In the transaction phase, the throughput of FVD is 77% higher than that of QCOW2 (by comparing the "FVD" bar and the "QCOW2 bar in the "virtio-partition" group of Figure 4(b)).

To understand the root cause of the performance difference, we perform a deep analysis for the results in the "virtio-partition" group of Figure 4(a). We run the *blktrace* tool in the host to monitor disk I/O activities. QCOW2 causes 45% more disk I/Os than FVD does, due to QCOW2's reads and writes to its metadata. Moreover, the average seek distance in QCOW2 is 5.6 times longer than that in FVD, due to QCOW2's VBA-IBA mismatching problem, as explained in Section 2.1.

5 Related Work

Despite the widespread use of VMs, there is no published research on how image formats impact disk I/O performance. Existing popular image formats (including QCOW2 [5], VDI [10], VMDK [11], and VHD [6]) all allocate storage space for a data block at the end of the image file when the block is written for the first time, regardless of the block's virtual address. This mismatch between VBA and IBA invalidates many optimizations in guest file systems, as discussed in Section 2.1. Moreover, they all unnecessarily mix the function of CoW dirty block tracking with the function of storage space allocation. This leads to doing storage allocation twice (first in the image format and then in the host file system), which causes data fragmentation twice and doubles the disk I/O overhead for metadata access.

Existing virtual disks support neither copy-on-read (CoR) nor adaptive prefetching. Some virtualization solutions do support CoR or prefetching, but they are implemented for specific use cases, e.g., virtual appliance [2] and VM migration [7]. By contrast, FVD provides CoR and prefetching as standard features of a virtual disk, which can be easily deployed in many different use cases. Moreover, those previous works use CoW and CoR but do not study how to optimize the CoW and CoR techniques themselves to reduce overhead.

Collective [2] provides desktop as a service across the Internet. It uses CoW and CoR to hide network latency. Its local disk cache makes no effort to preserve a linear mapping between VBA and IBA, and may cause a long disk seek distance as that in popular CoW image formats. Collective also performs adaptive prefetching. It halves the prefetch rate if a certain "percentage" of recent requests experience a high latency. Our evaluation shows that it is hard to set a proper "percentage" to reliably detect congestion. Because storage servers and disk controllers perform read-ahead in large chunks for sequential reads, a large percentage (e.g., 90%) of a VM's prefetching reads hit in the read-ahead caches and experience a low latency. When a storage server becomes busy, the "percentage" of requests that hit in the read-ahead caches may change little, but the response time of those cache-miss requests may increase dramatically. In other words, this "percentage" does not correlate well with the achieved disk I/O throughput.

6 Conclusion

FVD is a holistic virtual disk solution for both Cloud and non-Cloud environments. A design principle of FVD is to make all functions orthogonal so that each function can be enabled or disabled individually. The purpose is to support diverse use cases without being burdened with the overhead of all functions. Using copy-on-write, copy-on-read, and adaptive prefetching, FVD supports instant VM creation and instant VM migration, even if the VM image is stored on direct-attached storage. The source code of FVD is publicly available at `https://researcher.ibm.com/researcher/view_project.php?id=1852`.

References

[1] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX FREENIX Track*, 2005.

[2] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The Collective: A Cache-Based System Management Architecture. In *NSDI*, 2005.

[3] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[4] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[5] M. McLoughlin. The QCOW2 Image Format. `http://people.gnome.org/~markmc/qcow-image-format.html`.

[6] Microsoft VHD Image Format. `http://technet.microsoft.com/en-us/virtualserver/bb676673.aspx`.

[7] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *OSDI*, 2002.

[8] C. Tang. FVD: a High-Performance Virtual Machine Image Format for Cloud. This is the longer version of the USENIX'11 paper with the same title, available at `https://researcher.ibm.com/researcher/view_project.php?id=1852`.

[9] The IBM Cloud. `http://www.ibm.com/services/us/igs/cloud-development/`.

[10] VirtualBox VDI Image Format. `http://forums.virtualbox.org/viewtopic.php?t=8046`.

[11] VMware Virtual Disk Format 1.1. `http://www.vmware.com/technical-resources/interfaces/vmdk.html`.

Okeanos: Wasteless Journaling for Fast and Reliable Multistream Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis
Department of Computer Science
University of Ioannina, Greece

Abstract

Synchronous small writes play a critical role in the reliability and availability of file systems and applications that use them to safely log recent state modifications and quickly recover from failures. However, storage stacks usually enforce page-sized granularity in their data transfers from memory to disk. We experimentally show that subpage writes may lead to storage bandwidth waste and high disk latencies. To address the issue in a journaled file system, we propose *wasteless journaling* as a mount mode that coalesces synchronous concurrent small writes of data into full page-sized blocks before transferring them to the journal. Additionally, we propose *selective journaling* that automatically applies wasteless journaling on data writes whose size lies below a fixed preconfigured threshold. In the Okeanos prototype implementation that we developed, we use microbenchmarks and application-level workloads to show substantial improvements in write latency, transaction throughput and storage bandwidth requirements.

1 Introduction

Synchronous small writes lie in the critical path of several contemporary systems that target fast recovery from failures with low performance overhead during normal operation [1, 4, 5]. Typically, synchronous small writes are applied to a sequential file (*write-ahead log*) in order to record updates before the actual modification of the system state. In addition, the system periodically copies its entire state (*checkpoint*) to permanent storage. After a transient failure, recent state can be reconstructed by replaying the logged updates against the latest checkpoint. Write-ahead logging improves system reliability by preserving recent updates from failures; it also increases system availability by substantially reducing the subsequent recovery time. The method is widely applied in general-purpose file systems [6], relational databases [5], distributed key-value stores [4], event processing engines [3], and other mission-critical systems [7]. Furthermore, logging is useful for checkpointing parallel applications to preserve multiple hours or days of processing after an application or system crash [1].

Today, several file systems use a log file (*journal*) in order to temporarily move data or metadata from memory to disk at sequential throughput. Thus, they postpone the more costly writes to the file system without penalizing the corresponding latency perceived by the applications. A basic component across current operating systems is the page cache that temporarily stores recently accessed data and metadata in case they are reused soon. It receives byte-range requests from the applications, and communicates with the disk through page-sized blocks. The page-sized block granularity of disk accesses is prevalent across all data transfers, including data and metadata updates or the corresponding journaling whenever it is used. Asynchronous small writes improve their efficiency, when multiple consecutive requests are batched into page-sized blocks before they are flushed to disk. Instead, each synchronous write is flushed to disk individually causing data and metadata traffic of multiple full pages, even if the bytes actually modified across the pages collectively occupy much less space.

In Figure 1, we measure the amount of data written to the journal across different mount modes. We use a synthetic workload that consists of 100 concurrent threads with periodic synchronous writes of varying request sizes (one req/s). We include the ordered, writeback, and journal –refered to as data journaling from now on for clarity– modes of the ext3 file system (Section 4). As the request size increases up to 4KB, the traffic of data journaling remains almost unchanged at a disproportionately high value. At each write call, the mode appends to the journal the entire modified data and metadata blocks rather than only the corresponding block modifications. Instead, the ordered and writeback modes incur almost linearly increasing traffic, because they only store to the journal the blocks that contain modified metadata.

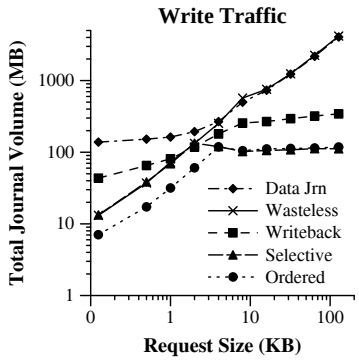


Figure 1: During a 5min interval, we measure the total write traffic to the journal device across different mount modes of the ext3 file system on Linux.

We set as objective to reduce the journal traffic so that we improve the performance of reliable storage at low cost. Thus, we introduce *wasteless journaling* and *selective journaling* as two new mount modes, that we propose, design and fully implement in the Linux ext3 file system. We are specifically concerned about highly concurrent multithreaded workloads that synchronously apply small writes over common storage devices [1, 4, 7]. We target to save the disk bandwidth that is currently wasted due to unnecessary writes of unmodified data, or writes with high positioning overhead. The operations in both these cases occupy valuable disk access time that should be used for useful data transfers instead. To achieve our goal we transform multiple random small writes into a single block append to the journal.

We summarize our contributions as follows: (i) Consider the reduction of journal bandwidth in current systems as a means to improve the performance of reliable storage at low cost; (ii) Design and fully implement wasteless and selective journaling as optional mount modes in a widely-used file system; (iii) Discuss the implications of our journaling optimizations to the consistency semantics; (iv) Apply micro-benchmarks, a storage workload and database logging traces over a single *journal* spindle to demonstrate performance improvements up to an order of magnitude across several metrics; (v) Use a parallel file system to show that wasteless journaling doubles, at reasonable cost, the throughput of parallel application checkpointing over small writes.

In the remaining paper, we summarize the related research in Section 2, present architectural aspects of our design in Section 3, while in Section 4 we describe the implementation of the Okeanos prototype system. In Section 5, we explain our experimentation environment, in Section 6 we present performance measurements across different workloads, and in Section 7 we outline our conclusions and future work.

2 Related Work

The log-structured file system addresses the synchronous metadata update problem and the small-write problem by batching data writes sequentially to a segmented log [9]. In transaction processing, group commit is a known database logging optimization that periodically flushes to the log multiple outstanding commit requests [5]. The above approaches gather multiple block writes into a single multi-block request instead of fitting multiple subpage modifications into a single block that we do. Also, subpage journaling of *metadata* updates is already available in commercial file systems, such as IBM JFS and MS NTFS [8]. Adding extra spindles to improve I/O parallelism or non-volatile RAM to absorb small writes could also reduce latency and raise throughput [6]. However, such solutions carry drawbacks that primarily have to do with increased cost and maintenance concerns.

A structured storage system may maintain numerous independent log files to facilitate load balancing in case of failure [4]. However, concurrent sequential writes to the same device create a random-access workload with low disk throughput. To address this issue, the system may store multiple logs into a single file and separate them by record sorting during recovery. Similarly, for the storage needs of parallel applications in high-performance computing, specialized file formats are used to manage as a single file the data streams generated by multiple processes [1]. Instead, we aim to handle the above cases at low cost through the mount modes that we add to a general-purpose file system.

The Echo distributed file system logged subpage updates for improved performance and availability, but bypassed logging for page-sized or larger writes [2]. However, Echo was discontinued in the early nineties partly because its hardware lacked fast enough computation relative to communication. Recent research introduced semantic trace playback to rapidly emulate alternative file system designs [8]. In that context, the authors emulated writing block modifications instead of entire blocks to the journal, but didn't consider the performance and recovery implications. Due to the obsolete hardware platform or the high emulation level at which they were applied, the above studies leave open the general architectural fit and actual performance benefit of journal bandwidth reduction in current systems.

3 System Design

We set as objective to safely store recent state updates on disk and ensure their fast recovery in case of failure. We also strive to serve the synchronous small writes and subsequent reads at sequential disk throughput with low bandwidth requirements. We are motivated by the important role that small writes play for reliable storage and the lack of comprehensive studies on subpage

data logging in current systems. In order to reduce the storage bandwidth consumed by data journaling, we designed and implemented a new mount mode that we call *wasteless journaling*. During synchronous writes, we transform partially modified data blocks into descriptor records that we subsequently accumulate into special journal blocks. We synchronously transfer all the data modifications from memory to the journal device. After timeout expiration or due to shortage of journal space, we move the partially or fully modified data blocks from memory to their final location in the file system.

With goal to reduce the journal I/O activity during sequential writes, we further evolved wasteless journaling into *selective journaling*. In that mount mode, the system automatically differentiates the write requests based on a fixed size threshold that we call *write threshold*. Depending on whether the write size is below the write threshold or not, we respectively transfer the synchronous writes to either the journal or directly the final disk location. Thus, we apply data journaling in only those cases that either multiple small writes can be coalesced into a single journal block according to wasteless journaling, or different data blocks that have been fully modified are scattered across multiple locations in the file system. We anticipate that journaling of the modified blocks will reduce the latency of synchronous writes through the sequential throughput offered by the journal device.

For consistency across system failures, each write operation delays metadata updates on disk, until the completion of the corresponding data updates. In wasteless journaling, we log both data and metadata into the journal to consider a write operation effectively completed. Synchronous writes from the same thread are added to the journal sequentially. In case of failure, a prefix of the operation sequence is recovered through the replay of the data modifications that have been successfully logged into the journal. Instead, selective journaling allows a synchronous write sequence to have a subset of the modified data added to the journal, and the rest of the modified data directly transferred to the final location in the file system. Given that a synchronous write from a single thread must be transferred to disk immediately, it only makes sense to accumulate into a journal block the writes from different concurrent threads. As a result, wasteless and selective journaling are mostly beneficial in concurrent environments with multiple writing streams that include frequent small writes.

In selective journaling, we call *update sequence* of a disk block a series of multiple incoming updates applied to the same block buffer. The updates don't have to be back-to-back, but there should be no in-between transfer of the respective buffer to the final disk location. If the first update in such a sequence has subpage size, we log to the journal the entire update sequence of this buffer. Thus, we handle consistency in a relatively clean way, because we eliminate the case that we turn off the journaling of a particular buffer halfway through a transaction. On the other hand, if the first update of the buffer is page-sized, we decide to skip journaling for the entire update sequence of the corresponding block. In our experience, the above two transitions in update sizes along a sequence occur infrequently. Therefore, we anticipate low impact to the journaling activity of selective journaling.

4 The Okeanos Prototype Implementation

We implemented wasteless and selective journaling in the Okeanos prototype that we developed based on Linux ext3. Originally, ext3 first copies the modified blocks into the journal, then transfers them to their final disk location. In *data journaling* mount mode, data and metadata blocks are copied to the journal, before they update the file system. To reduce the risk of data corruption, the *ordered* mode only copies the metadata blocks to the journal, after the associated data blocks have updated the file system. The *writeback* mode copies only metadata blocks to the journal, without any constraint in the relative order of data and metadata updates to the file system. It is the weakest mode in terms of consistency and we don't consider it any further in the rest of the paper.

The Linux kernel uses the *page cache* to keep in memory data and metadata of recently accessed disk files. For every disk block cached in memory, a *block buffer* stores its data and a *buffer head* maintains the related bookkeeping information. The page cache manages disk blocks in page-sized groups called *buffer pages*. We use block and page interchangeably, because they typically have the same size. Ext3 implements the journal as either a hidden file in the file system or a separate disk partition. Each *log record* in the journal contains an entire modified block instead of the byte range actually affected. However, the system only needs to log the updated part of each modified block and merge it into the original block to get its latest version during a recovery. To achieve that, we introduce a new type of journal block that we call *multiwrite block*. We only use multiwrite blocks to accumulate the updates from *data* writes that partially modify block buffers. When a block buffer contains metadata or is fully modified by a write operation, we can send it directly to the journal without the need to create an extra copy first in the page cache. We call *regular block* such a journal block.

When a write request of arbitrary size enters the kernel, the request is broken into variable-sized updates of individual block buffers. In wasteless journaling, if the size of a buffer update is less than the block size, we copy the corresponding data modification into a multiwrite block. Otherwise, we point to the entire modified

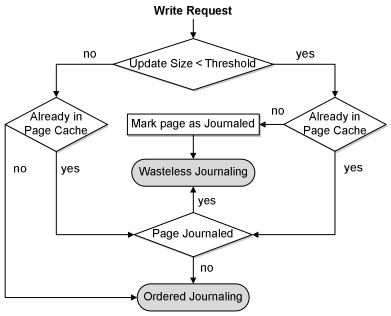
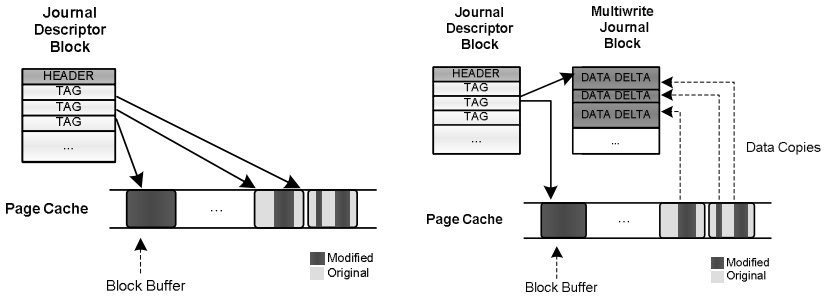


Figure 2: Alternative execution paths of a write request in the selective journaling mode.



(a) (b)

Figure 3: (a). In data journaling, the system sends to the journal the entire blocks modified by write operations. (b) In wasteless journaling, we accumulate multiple data writes into a single multiwrite journal block.

block in the page cache. For selective journaling, we have the write threshold fixed to the page size of 4KB. When a buffer update has size smaller than the write threshold, then we mark the corresponding page as *journaled*. Correspondingly, we copy the modification to the multiwrite block. We clear the journaled flag, after we transfer the corresponding block to its final location on disk. In Figure 2, we use a flowchart to summarize the possible execution paths of a write request through selective journaling.

A system call may consist of multiple low-level operations that atomically manipulate disk data structures of the file system. For improved efficiency, the system groups the records of multiple calls into one *transaction*. Before the transaction moves to the commit state, the kernel allocates a *journal descriptor block* with a list of *tags* that map block buffers to their final disk location. For each block buffer that will be written to the journal, the kernel allocates an extra buffer head specifically for the needs of journaling I/O. Additionally, it creates a *journal head* structure to associate the block buffer with the respective transaction. For writes that only modify part of a block, we expanded the journal head with two extra fields that contain the offset and the length of the multiwrite block pointed to by the buffer head (Figure 3). When we start a new transaction, we allocate a journal descriptor block that contains multiple fixed-length tags, one per write. In our system, we introduce three new fields in each tag: (i) a flag to indicate the use of a multiwrite block, (ii) the length of the write in the multiwrite block, and (iii) the starting offset of the modification in the final data block.

A transaction is *committed*, if it has flushed all its records to the journal; it is *checkpointed*, if all the blocks of a committed transaction have been moved to their final location on disk and the corresponding log records are removed from the journal. If the journal contains log records after a crash, the system initiates a recovery process during which we retrieve the modified blocks from the journal. In the case of multiwrite blocks, we apply the updates to blocks that we read from the corresponding final disk locations. We read into memory and update the appropriate block, as specified by the final disk location and the starting offset in the tag. However, if the multiwrite flag is not set, then we read the next block of the journal and treat it as a regular block. We write every regular block directly to the final disk location without need to read first its older version from the disk.

Both data and wasteless journaling guarantee the atomicity of updates, because they replay the modifications of the committed transactions until they fully reach the file system. Instead, selective journaling makes a decision whether to journal or not an update sequence based on the size of the first write. Journaling of an update sequence implies atomicity of the modification for the corresponding block, while direct transfer of the block to the file system implies consistency similar to that of ordered mode.

5 Experimentation Environment

We developed the Okeanos prototype implementation of wasteless and selective journaling by modifying 684 lines of code across 19 files of the original Linux kernel version 2.6.18. Members of our team used the prototype system as working environment for several months. In our experiments we use x86-based servers with one quad-core 2.66GHz processor, 3GB RAM, two Seagate Cheetah SAS 300GB 15KRPM disks and one active gigabit ethernet port. Unless we specify differently, we assume synchronous write operations with the journal and data partition on two separate disks. Our results have half-length of 90% confidence interval within 10% of the reported average. We flushed the page cache between all the repetitions of our experiments.

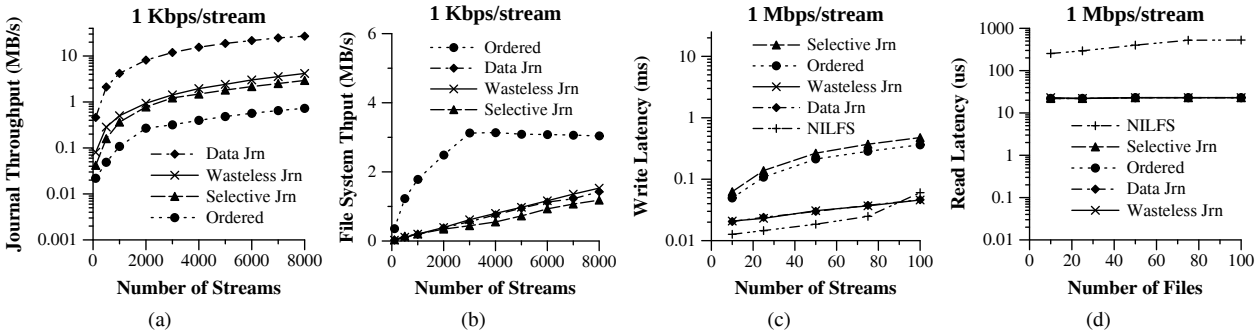


Figure 4: (a) At 1Kbps, the journal throughput (lower is better) of selective and wasteless journaling lies approaches that of ordered, unlike data journaling which is several factors higher. (b) In comparison to ordered at 1Kbps, the remaining three modes reduce file system throughput by several factors (lower is better). (c) At 1Mbps, the selective and ordered modes incur much higher latency in comparison to the other ext3 modes or NILFS. (d) If we create multiple files concurrently, read requests of 4KB with NILFS take an order of magnitude longer with respect to ext3.

6 Performance Evaluation

First, we produce a random I/O traffic by running a number of concurrent threads directly on the file server. Each thread appends data to a separate file by calling one synchronous write per second. At increasing number of 1Kbps streams, Figure 4(a) shows that the journal throughput of data journaling is an order of magnitude higher than that of the other modes (up to 27MB/s). On the contrary, selective and wasteless journaling limit the traffic up to about 4MB/s. In Figure 4(b), we measure the write throughput of the file system device. The ordered mode wastes disk bandwidth by sending each write to the final location in units of 4KB. Instead, wasteless, selective and data journaling leave dirty pages temporarily in memory before coalescing them into the file system. With controlled system crashes, we additionally found that selective and wasteless journaling tend to reduce the recovery time of data journaling (by more than 20% in some cases).

Next, we examine the average latency of synchronous writes. In Figure 4(c) with 1Mbps streams, ordered and selective incur orders of magnitude higher latency than the other modes. At 1Kbps (not shown), selective tends to become identical to wasteless journaling. In asynchronous writes that we also tried, we found selective and wasteless journaling to reduce the latency of ordered and data journaling up to two orders of magnitude. In Figure 4(c), we also consider a stable Linux port (NILFS) of the log-structured file system [9]. The write latency of NILFS is comparable to that of wasteless and data journaling. In Figure 4(d), we use a thread to read sequentially one after the other different numbers of files that we previously created concurrently at 1Mbps each, using NILFS or ext3. Then, we measure the average time to read a 4KB block. We observe that NILFS is an order of magnitude slower with respect to ext3. In fact, NILFS interleaves the writes from different files on disk, which may lead to poor storage locality during sequential reads.

We use the Postmark benchmark to examine the performance of small writes as seen in electronic mail, netnews and web-based commerce. We apply version 1.5 with synchronous writes added by FSL of Stony Brook Univ. We assume an initial set of 500 files and use 100 threads for a total workload of 10,000 mixed transactions. We draw the file sizes from the default range, while I/O request sizes lie between 128 bytes and 128KB. In Figure 5(a), we observe that the transaction rate of wasteless journaling gets as high as 738tps. Across different request sizes, wasteless journaling consistently remains faster than the other modes, including data journaling. Instead, selective journaling lies between data journaling and ordered mode, which are slower than wasteless.

We also examine the OLTP performance benchmark TPC-C as implemented in Test 2 of the Database Test Suite. We used the MySQL open-source database system with the default InnoDB storage engine. We tested a configuration with 20 warehouses and 20 connections, 10 terminals per warehouse and 500s duration. InnoDB supports three methods for flushing the database transaction log to disk. In the default method 1 (*Cmt/Disk*), the log is flushed directly to disk at each transaction commit. In method 0 (*Prd/Disk*), the transaction log is written to the page cache and flushed to disk periodically. Finally, in method 2 (*Cmt/Cache*), the transaction log is written to the page cache at each transaction commit and periodically flushed to disk. During an execution of TPC-C, we collect a system-call trace of the MySQL transaction log. Subsequently, we replay a varied number of concurrent instances of the log trace over the ordered and wasteless journaling. In Figure 5(a), we see that wasteless journaling takes up to tens of seconds to complete each log

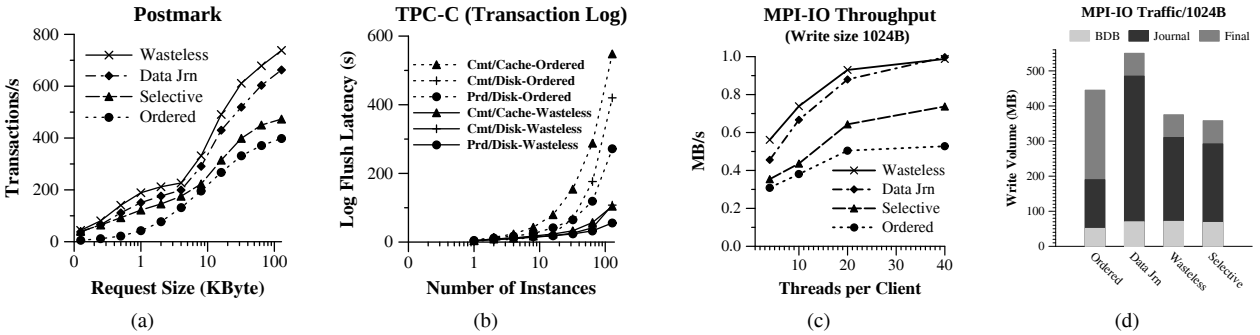


Figure 5: (a) Wasteless journaling consistently achieves the highest transaction rate in Postmark. (b) Across the three flushing methods of MySQL/InnoDB, wasteless journaling substantially reduces the latency to flush the transaction log to disk. (c) Wasteless journaling almost doubles the data throughput (higher is better) of ordered mode. (d) We measure the write traffic (lower is better) to BerkeleyDB (BDB), the journal (Journal) and the file system (Final) at the data server of PVFS2. Selective and wasteless journaling incur less traffic under the MPI-IO benchmark.

flush across the three methods of InnoDB at high load. Instead, ordered mode takes hundreds of seconds, as the number of instances approaches or exceeds 64.

Finally, we use our mount modes in the storage server of a PVFS2 multi-tier configuration. In a networked cluster, we use thirteen machines as clients, one machine as PVFS2 data server and one as PVFS2 metadata server. By default, each server uses a local BerkeleyDB database to maintain local metadata. At the data server we placed the BerkeleyDB on one partition of the root disk, and dedicated the entire second disk to the user data (file system and journal). We fixed the BerkeleyDB partition to ordered mode and tried alternative mount modes at the data disk. We enabled data and metadata synchronization, as suggested to avoid write losses at server failures. We used the LANL MPI-IO Test to generate a synthetic parallel I/O workload on top of PVFS2. In our configuration each process writes to a separate unique file, as suggested for best performance [1]. We varied between 4 and 40 the number of processes on each of the thirteen quad-core clients leading to total processes between 52 and 520. We tried 65000 writes of size 1024 bytes. In Figure 5(c), wasteless journaling almost doubles the throughput of ordered mode, while data journaling and selective lie between the other two modes. In Figure 5(d), wasteless journaling reduces by 42% the journal traffic of data journaling, while selective journaling further reduces the write volume of wasteless journaling.

7 Conclusions and Future Work

We rely on journaling of data updates in a file system to ensure their safe transfer to disk at low latency and high throughput without storage bandwidth waste. We design and implement a mount mode that we call wasteless journaling to merge into page-size blocks concurrent subpage writes to the journal. Additionally, we develop the selective journaling mode that only logs updates below a write threshold and transfers the rest directly to the file system. We experimentally demonstrate reduced write latency, improved transaction throughput with low journal bandwidth requirements. Our plans for future work include extension of our journaling methods for virtualization environments and solid-state disks.

8 Acknowledgments

We are thankful to our shepherd Junfeng Yang for his valuable guidance. In part supported by project INTERSAFE (No. 303090/YD7631) of the INTERREG IIIA program co-funded by EU and the Greek State.

References

[1] BENT, J., GIBSON, G., GRIDER, G., MCCLELLAND, B., NOWOCZYNSKI, P., NUNEZ, J., POLTE, M., AND WINGATE, M. Plfs: a checkpoint filesystem for parallel applications. In *ACM/IEEE SC* (2009), pp. 1–12.

[2] BIRRELL, A. D., HISGEN, A., JERIAN, C., MANN, T., AND SWART, G. The echo distributed file system. Tech. Rep. TR-111, DEC Systems Research Center, Palo Alto, CA, Sept. 1993.

[3] BRITO, A., FETZER, C., AND FELBER, P. Minimizing latency in fault-tolerant distributed stream processing systems. In *IEEE ICDCS* (Montreal, QC, 2009), pp. 173–182.

[4] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *USENIX OSDI* (2006), pp. 205–218.

[5] GRAY, J., AND REUTER, A. *Transaction Processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993, ch. 9. Log Manager.

[6] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *USENIX Winter Technical Conference* (San Francisco, CA, Jan. 1994), pp. 235–246.

[7] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *USENIX OSDI* (Seattle, WA, 2006), pp. 1–14.

[8] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *USENIX ATC* (Anaheim, CA, 2005), pp. 105–120.

[9] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 26–52.

Toward Online Testing of Federated and Heterogeneous Distributed Systems

Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić
School of Computer and Communication Sciences, EPFL, Switzerland
email: firstname.lastname@epfl.ch

Abstract

Making distributed systems reliable is notoriously difficult. It is even more difficult to achieve high reliability for federated and heterogeneous systems, *i.e.*, those that are operated by multiple administrative entities and have numerous inter-operable implementations. A prime example of such a system is the Internet's inter-domain routing, today based on BGP.

We argue that system reliability should be improved by proactively identifying potential faults using an online testing functionality. We propose DiCE, an approach that continuously and automatically explores the system behavior, to check whether the system deviates from its desired behavior. DiCE orchestrates the exploration of relevant system behaviors by subjecting system nodes to many possible inputs that exercise node actions. DiCE starts exploring from current, live system state, and operates in isolation from the deployed system. We describe our experience in integrating DiCE with an open-source BGP router. We evaluate the prototype's ability to quickly detect origin misconfiguration, a recurring operator mistake that causes Internet-wide outages. We also quantify DiCE's overhead and find it to have marginal impact on system performance.

1 Introduction

Internet's inter-domain routing, based on the standard Border Gateway Protocol (BGP), is a prime example of a distributed system that is fundamentally *federated* and *heterogeneous*. Multiple administrative domains autonomously control their own BGP routers and policies, while ensuring universal connectivity. Further, the open standards upon which the Internet is built allow for and promote numerous, mutually inter-operating implementations of BGP. Examples of other systems of such nature include DNS, electronic mail, peer-to-peer content distribution [10], content and resource peering [5].

Recent events have shown that Internet's routing falls short of ensuring reliable operation at all times. For example, Pakistan Telecom mistakenly managed to hijack the vast majority of traffic directed toward YouTube, making the popular website unreachable to many users for almost two hours [2][1].

In general, system behavior is the aggregate result of interleaved actions of system nodes, each of which is generally driven by code as well as configuration. Understandably, it is hard to reason *a priori* about every corner case and anticipate all possible combinations of system configurations. As a consequence, insidious bugs can survive until the system is deployed or configuration mistakes become a problem under certain unanticipated conditions — all these with dire consequences for the system's reliable operation.

We argued [9] that making heterogeneous and federated distributed systems reliable is challenging because (i) the source code of every node may not be readily available for testing and (ii) competitive concerns are likely to induce individual providers to keep private much of their current state and configuration.

Our overarching vision is to harness the continuous increases in available computational power and bandwidth to improve the reliability of distributed systems. In particular, we argue for an online testing functionality that strives to detect what node actions lead to potential faults (*i.e.*, deviations of system components from their expected behavior).

We have to address several difficult challenges (of which a more thorough account is in [9]). First, the federated nature of the systems we target give rise to a number of issues because of the different administrative domains desire to keep private their node states and configurations. Most importantly, no single node can have unrestricted access to remote node state and configuration.

[1] This problem persists to this day. China Telecom managed to hijack 10% of the Internet prefixes as recently as April 2010. Google's services were mistakenly hijacked in July and August of 2010 [1].

This affects how we can drive the exploration of system behavior and how we can check system-wide state to detect faults. This also hinders the possibility of simply applying existing approaches that drive exploration from the initial state (*e.g.*, [22]). In addition, we need to carefully consider what information crosses domain boundaries and how to preserve its confidentiality. Second, system heterogeneity makes it difficult if not impossible to have local access at one node to the source or binary code of other nodes. This difficulty and other constraints we mentioned above mean that we cannot use existing techniques for live model checking (*e.g.*, [21]) that operate locally. Last but not least, systematic exploration of system behavior or even single node behavior runs into the problem of exponential explosion of the number of possible node actions.

In this paper, we introduce DiCE, an approach that continuously and automatically explores the system behavior, to check whether the system deviates from its desired behavior. DiCE orchestrates the exploration of relevant system behaviors by subjecting system nodes to many possible inputs that systematically exercise node actions. To quickly reach relevant states and overcome problems with an exponential number of actions, DiCE starts exploring from current system state, while it operates in isolation from the deployed system.

We describe our general vision and outline the problem we want to address in DiCE (§2.1). We provide an initial design (§2.3) and discuss our experience in integrating DiCE with a BGP router (§3). We evaluate (§4) the prototype's ability to quickly detect origin misconfiguration, a recurring operator mistake that causes Internet-wide outages. We also quantify DiCE's overhead and find it to have marginal impact on system performance.

2 DiCE

We start by providing an overview of the problem that we want to address.

2.1 Problem overview

Our goal is to systematically explore system behavior so as to detect potential faults. At the same time, the approach that orchestrates the exploration of system behavior needs to accommodate the constraints of federated and heterogeneous environments.

A central question for reaching this goal is in understanding how to drive system behavior. We observe that distributed system behavior is the aggregate result of interleaved node actions. In turn, these actions are determined by the paths taken through the code running at the nodes that is driven by the configuration and the inputs. Therefore, to explore node actions, we want to subject the node's code to inputs that systematically exercise the node's possible actions. In other words, we need a mechanism that systematically exercises the node's code paths.

In practice, achieving extensive path coverage is greatly limited by the exponential explosion in the number of possible code paths. Given our desire to quickly detect potential faults, we would ideally just focus on covering relevant states. However, these are usually deep in the execution path. Recall that we target systems that are likely to run for a long time over which a large history of inputs accumulates. Thus, we need to avoid the need to replay a long history of inputs from initial state to reach a desired point in the code, as doing so can be prohibitively time-consuming.

An intriguing question is whether local testing of a single node is sufficient to detect such actions. While local testing is certainly a necessary step, we argue that it alone is not sufficient. In fact, local testing does not allow to observe far reaching consequences of single node actions. These consequences need to be observed from a system-wide perspective.

Therefore, we still need to be able to judge the system-wide consequences of node actions. In the general case, this cannot be done locally because a node does not know, and we assume cannot obtain the state and configuration, of other nodes. Also, a remote node could be running a different implementation. Effectively, we face the problem of how to let the local node communicate with remote nodes during exploration of behavior. However, we must not affect the deployed system and, at the same time, we need to support checking node states while preserving confidentiality of private information.

In summary, we are trying to address these questions: (i) how can we automatically exercise code paths? (ii) how can we enable code path exploration to guide state space exploration? (iii) how can we extend the horizon of local state space exploration to reach across the network? and (iv) how can we check for faults while preserving privacy between parties?

In this paper, we focus on the first two of these questions and provide a discussion around the other two. Before presenting our initial design, we proceed to briefly review certain software testing techniques that offer the basic mechanics necessary for systematically exploring a node's code paths.

2.2 Background

Symbolic execution (*e.g.*, see [8, 13]) is an automated testing technique that executes a program by treating the inputs to the program as symbolic. Upon encountering a branch that involves symbolic values, the symbolic execution engine creates the constraints that correspond to

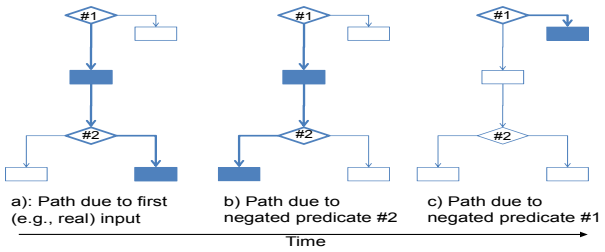


Figure 1: A concolic execution engine negates the predicates to try to systematically explore code paths (blocks that are executed in each run are shown as shaded).

both sides of the branch, and schedules execution to take place down both paths. It then queries a constraint solver to determine which paths are feasible, so that they can be explored. While symbolic execution is in theory capable of exploring all possible paths in the program, in practice it is severely limited as the number of paths to explore in an application grows exponentially with the size of the input and the number of branches in the code. A typical symbolic execution engine starts exploring paths from the beginning of the program and progressively explores all paths for which it can find suitable input values.

Concolic execution (*e.g.*, see [7, 11]) is a variant of symbolic execution that, instead of strictly operating on symbolic inputs, executes the code with concrete inputs while still collecting constraints along code paths. To drive execution down a particular path, the concolic execution engine picks a constraint (*e.g.*, branch predicate) and queries the constraint solver to find a concrete input value that negates the constraint. Figure 1 illustrates this process. The main benefit of concolic execution is the ease in interacting with the environment (due to the use of concrete values), and less overhead during execution than the "classic" symbolic execution (*e.g.*, only one call to the constraint solver for every branch).

2.3 Initial design

DiCE employs a concolic execution engine to solve the mechanical problem of exercising all possible code paths. Unlike standard concolic execution, DiCE starts exploring from the current, live state because of the desire to (i) quickly detect potential faults, and (ii) avoid the overhead of replaying execution from initial state to reach a desired point in the code (as we expect a large history of inputs).

First, DiCE takes a node checkpoint. Then, DiCE clones this checkpoint and feeds it with a previously observed input (*i.e.*, a message) to record the constraints that are encountered on the code path executed by invoking a message handler with that input. We rely on the programmer to identify message handlers and we only use those to process inputs for path exploration. This design decision lets us quickly zoom in on the relevant code, at the expense of requiring some developer involvement[2]. After completing the recording of these initial constraints recording, the concolic execution engine starts negating constraints one at a time, resulting in a set of inputs. To explore a particular input, DiCE makes a clone of the checkpoint, and then resumes execution with that input from the checkpointed state. The constraints encountered on the code path during execution with that particular input are once again recorded and used to update the aggregate set of constraints so far encountered. Updating the aggregate set is important for achieving full coverage, since the previous runs might not have reached all branches that exist in the code.

Note that we want the exploratory execution over a node checkpoint to work alongside the running system. Therefore, DiCE intercepts the messages generated during exploration.

2.4 Discussion

We consider the ability to explore node actions that we described as the initial building block for providing a full online testing functionality.

In fact, once we can locally exercise all possible node actions, we can then turn to how to observe their consequences on the system-wide state. We anticipate that we would let these actions result into messaging with other nodes in a way that doesn't affect the live system. For instance, we could intercept all messages and let them go through isolated communication channels. In addition, we would enable remote nodes to checkpoint their state and process these messages in isolation over their checkpointed states. Effectively, this would extend the scope of the concolic execution engine to reach across the network and exercise system behavior.

Ultimately, we want to check system-wide states for faults. We envision that, by having a notion of desired system behavior, we could check whether the system deviates from its desired behavior during the exploration with particular inputs. However, it is challenging to check system-wide states without compromising the confidentiality of private information. As we noted earlier, there cannot be unrestricted access to node states and configurations. In essence, we would want to control the information shared across domains and ensure that nodes only communicate state information through a narrow interface yet capable to allow us to detect faults.

[2]Given the great importance of the deployed federated systems, this effort is well-justified.

3 Experiences with the BGP use case

This section describes our DiCE prototype and details our experiences for integrating with the BGP implementation of BIRD [3] 1.1.7 open-source routing daemon. Because of space limit, we omit a review of BGP and point the reader to [14] for a succinct overview and to the RFC [20] for full details. We first introduce Oasis [11], the concolic execution engine we use.

3.1 Oasis

We use the Oasis concolic execution engine [11] as the basis for code path exploration. Oasis is a result of substantial modification of the Crest [7] concolic execution engine. Oasis instruments C programs using CIL [18] to be able to track at run-time the statements executed and record the constraints on symbolic inputs. Oasis handles the entire C language and supports interaction with the network and filesystem. Oasis has multiple search strategies, and it can execute multiple explorations in parallel. The default exploration strategy, which we use, attempts to cover all execution paths reachable by the set of controlled symbolic inputs.

3.2 Prototype implementation

Our DiCE prototype consists of a modified version of Oasis and a part written in C and integrated in BIRD.

We modify Oasis in three ways. First, we introduce support in Oasis to explore by resuming execution from a checkpoint instead of starting a new execution for each set of inputs. Second, we change the Oasis filesystem/network model to control the interactions of the program under test with the environment and ensure isolation from the running system. Third, we modify Oasis to allow both the original and the instrumented code to be automatically compiled and linked in a single executable, where they co-exist and operate on the same data in a similar fashion to the work by Anagnostakis *et al.* [6]. This enables the running systems to run with virtually no overhead, and only during exploration, which takes place off the critical path, switch to the instrumented code.

For integrating with BIRD, we first change its BGP implementation to mark certain inputs as symbolic. We choose to treat UPDATE messages as the basis to derive new inputs during exploration. In BGP, UPDATE messages are the main drivers for state change while the other state changing messages are only responsible for establishing or tearing down peerings and we leave them for future work.

A simple approach would be to mark an entire UPDATE message as symbolic. However, this has the effect of causing Oasis to produce a large variety of invalid messages that simply exercise the message parsing code[3]. This is undesirable for us because we want to explore node actions, and so we need to go deeper in the message processing code. As the format of BGP messages is well-defined in the RFC [20], we selectively define as symbolic small-sized inputs that directly derive from the message. For instance, the Network Layer Reachability Info (NLRI) region of the message contains the announced routes with their respective netmask lengths. We mark these as symbolic. An UPDATE message also typically carries multiple path attributes each of which is encoded as a type, length, and value fields that can also be marked symbolic. However, one needs to be careful that the symbolic length matches the actual length of the value field and that its semantic is consistent with the attribute type; otherwise the message is invalid and of no utility. Note that this approach is very effective in reducing the space of exploration because the produced messages are always syntactically valid.

In practice, this choice allows DiCE to construct inputs that exercise BGP behavior in two dimensions: the first due to BIRD's code implementing BGP, the second as the result of the particular configuration currently in use. This is because the source code instrumentation encompasses the BIRD's configuration interpreter and so allows Oasis to record constraints for the interpreted configuration. Therefore, the explored execution paths are comprehensive of both code and configuration. Finally, to enable Oasis to perform path exploration of BIRD's code, we handle the well-known cases that are difficult or impossible to handle in symbolic execution. For example, we avoid recording constraints that result from applying hash functions, as they cannot be reversed.

We make a second change to BIRD for taking a checkpoint. We implement this procedure by simply using the `fork` system call. This way of checkpointing allows us to create a large number of checkpoints with a small memory footprint. We are careful to isolate the forked process from its parent by closing the open sockets.

4 Evaluation

We use a 2.6 GHz 48-core machine with 64 GB of RAM, running Linux 2.6.30. Using virtual interfaces and multiple BIRD router instances, we install the 3-router topology shown in Figure 2. DiCE runs in the Provider's router. The DiCE-enabled router loads 319,355 prefixes from the "rest of the Internet" where we replay a BGP trace obtained from RouteViews (a full dump plus 15-min updates trace from route-views.eqix at April 1, 2010, 17:28 UTC). To be able to detect route leaks, we con-

[3] Which ought to be correct and could anyway be tested with a symbolic execution engine that targets single-machine code.

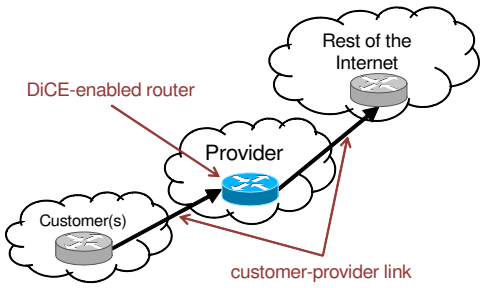


Figure 2: The experimental topology.

figure a partially correct route filtering. Customer route filtering happens in the provider and is a best common practice currently adopted by several large ISPs to defend against BGP prefix hijacking.

4.1 Performance impact

Here we provide a summary of the micro-benchmarks we run to understand how much DiCE impacts memory and CPU usage during exploration.

Memory overhead. We perform measurements that quantify the memory overhead on a BIRD router that has a full routing table loaded. We then run the exploration while the routers processing a 15 minute trace replay of BGP messages. The checkpoint process has 3.45% unique memory pages. The processes forked for exploring from the checkpoint process consume on average 36.93% pages more (maximum of 39%).

CPU/performance. We use the number of BGP update messages the DiCE-enabled router handles per second as a measure of how much the performance is affected while running exploration. The BIRD processes are configured to run on separate CPU cores, with the explorer having to share the single CPU core with its checkpoints that are used during exploration.

Under full load (running the exploration while loading the routing table), the BIRD process manages 13.9 updates per second. Without exploration, in the same interval of time during the trace replay, it is handling 15.1 updates per second. Thus, the performance impact even in this most stressful case is still small, namely 8%.

In a different, more realistic scenario, we run the exploration a few minutes inside the replay of a real-time trace of 15 min (after the full routing table was loaded). In this case the difference is negligible, with the BIRD process managing 0.272 queries per second during exploration and 0.287 when free to use the full CPU core.

4.2 Detecting route leaks

In a highly publicized router misconfiguration incident, Pakistan Telecom (an ISP) managed to divert to itself and drop the vast majority of traffic directed toward YouTube, the popular video-on-demand website. Consequently, this important service was unavailable for almost two hours [2]. In this particular case of BGP misconfiguration called prefix hijacking, there were two compounded errors that caused the fault. First, Pakistan Telecom announced a route that it had only intended to blackhole (*i.e.*, make YouTube unavailable to Pakistani residents). Second, PCCW, the upstream provider for Pakistan Telecom, did not have filters installed to limit the spreading of this announcement.

To replicate the IP prefix hijacking problem in our testbed, we misconfigured customer route filtering at the Provider AS. That is, its policy either fails to filter customer routes or has erroneous filters. Then, DiCE locally exercises all possible execution paths, which also include the "if" statements in the configured filters. For each exploratory message, we check whether the announced route (as determined by Oasis's manipulation of the NLRI) is accepted, and in this case we detect a potential hijack if that route overrides the origin AS of a route already in the routing table prior to starting exploration[4]. Certain prefixes are "hijackable" by nature, *e.g.*, those used for IP anycast, and would appear as false positives. DiCE can simply filter these out once it is made aware of the IP anycast address space.

The benefit from running DiCE for a network operator is significant, as DiCE clearly states which prefix ranges can be leaked. In the case of YouTube hijacking, Pakistan's upstream provider would have been able to install a correct filter.

5 Related work

CrystalBall [21], and MODIST [22] represent the state-of-the-art in model checking distributed system implementations. CrystalBall [21] proactively predicts inconsistencies that can occur in a running distributed system due to unknown programming errors, and effectively prevents them. MODIST [22] is capable of model checking unmodified distributed systems.

Symbolic [8, 13] and concolic execution [11, 7] are techniques for achieving complete coverage of possible code paths and are effective in discovering bugs for single-machine code. In DiCE, we leverage concolic execution as the base mechanism for exploring distributed system states starting from covering possible node actions and ultimately judge their system-wide impact.

Collaboration among a program's or a system's stakeholders has been used in similar contexts. For example, Liblit *et al.* [15] proposed an approach that infers bugs by gathering information from the program's users. A

[4]This assumes that the existing routes are trustworthy.

sampling technique is used to maintain a low instrumentation overhead. Orso *et al.* [19] suggested an approach for continuously analyzing deployed software with minimal instrumentation with the goal of improving software quality. In the context of collaborative security, Locasto *et al.* [16] proposed that members of an application community share the burden of monitoring for software flaws and attacks, and notify the rest of the community when such are detected.

Nagaraja *et al.* [17] focused on operator mistakes in Internet services and argued for the creation of an online validation environment to be used to check operator actions before they are made visible. We consider their approach complementary to DiCE, in that our approach could be extended to explore system behavior under specific operator actions before they are introduced in the running system. In the case of a single ISP's routers, Alimi *et al.* [4] proposed to install an alternative configuration with which network operators can test proposed changes before committing them to the production network. Feamster *et al.* [12] demonstrated the effectiveness of static analysis to look for faults in the set of router configurations, but cannot check live node states.

6 Conclusions

In this paper we argued for leveraging the increases in computational power and bandwidth to make federated and heterogeneous distributed systems more reliable. We presented a preliminary design of DiCE, a system that systematically exercises node behavior with the goal of ultimately checking the system-wide impact of each node behavior. We described our experience in integrating a path exploration engine with an open-source BGP router written in C. We also outlined the challenges in extending this kind of online testing to reach across the network. Finally, we demonstrated our prototype's ability to detect BGP route leaks - an important class of configuration errors that plagues the Internet.

Acknowledgments.

We thank the anonymous reviewers and our shepherd, Angelos D. Keromytis, for their helpful comments and suggestions. We are grateful to Jennifer Rexford, Katerina Argyraki and Jon Crowcroft for their feedback on earlier drafts of this work. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110 and the Hasler foundation (grant 2103).

References

[1] Google's services redirected to Romania and Austria. `http://bgpmon.net/blog/?p=314`.

[2] Pakistan hijacks YouTube. `http://www.renesys.com/blog/2008/02/pakistan_hijacks_youtube_1.shtml`.

[3] The BIRD Internet Routing Daemon. `http://bird.network.cz`.

[4] R. Alimi, Y. Wang, and Y. R. Yang. Shadow Configuration as a Network Management Primitive. In *SIGCOMM*, 2008.

[5] L. Amini, A. Shaikh, and H. Schulzrinne. Effective Peering for Multi-provider Content Delivery Services. In *INFOCOM*, 2004.

[6] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. In *USENIX Security Symposium*, 2005.

[7] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[8] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[9] M. Canini, D. Novaković, V. Jovanović, and D. Kostić. Fault Prediction in Distributed Systems Gone Wild. In *LADIS*, 2010.

[10] B. Cohen. Incentives Build Robustness in BitTorrent. In *P2PECON*, 2003.

[11] O. Crameri, R. Bachwani, T. Brecht, R. Bianchini, D. Kostić, and W. Zwaenepoel. Oasis: Concolic Execution Driven by Test Suites and Code Modifications. Technical Report LABOS-REPORT-2009-002, EPFL, 2009.

[12] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI*, 2005.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI*, 2005.

[14] A. Haeberlen, I. Avramopoulos, J. Rexford, and P. Druschel. NetReview: Detecting when Interdomain Routing Goes Wrong. In *NSDI*, 2009.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug Isolation via Remote Program Sampling. In *PLDI*, 2003.

[16] M. E. Locasto, S. Sidiroglou, and A. D. Keromytis. Software Self-Healing Using Collaborative Application Communities. In *SNDSS*, 2006.

[17] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *OSDI*, 2004.

[18] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Compiler Construction*, 2002.

[19] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software after Deployment. In *ISSTA*, 2002.

[20] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4) IETF RFC 4271. 2006.

[21] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[22] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

CDE: Using System Call Interposition to Automatically Create Portable Software Packages

Philip J. Guo and Dawson Engler
Stanford University

Abstract

It can be painfully hard to take software that runs on one person's machine and get it to run on another machine. Online forums and mailing lists are filled with discussions of users' troubles with compiling, installing, and configuring software and their myriad of dependencies. To eliminate this dependency problem, we created a system called CDE that uses system call interposition to monitor the execution of x86-Linux programs and package up the Code, Data, and Environment required to run them on other x86-Linux machines. Creating a CDE package is completely automatic, and running programs within a package requires no installation, configuration, or root permissions. Hundreds of people in both academia and industry have used CDE to distribute software, demo prototypes, make their scientific experiments reproducible, run software natively on older Linux distributions, and deploy experiments to compute clusters.

1 Introduction

Most programmers want other people to run their software. Unfortunately, the path from having a piece of software running on the programmer's own machine to getting it running on someone else's machine is fraught with potential pitfalls. For instance, the programmer might have forgotten to document a crucial step in the magic incantation needed during the installation process. Or forgotten to list a library version dependency, leading to mysterious run-time errors when the wrong version gets silently run on the user's machine. Or listed the right library version, but one which is either hard to obtain or conflicts with a library needed by a different program on the user's machine. Or the software itself might require libraries that depend on many other libraries, which themselves need to be transitively obtained and installed by the user, leading to an aggravating experience known as *dependency hell*. Finally, the user might lack the permissions or willingness to risk installing software packages as root in the first place, a common occurrence on corporate machines and clusters administered by IT staff.

To alleviate these frustrations, we have created an open-source tool named CDE that monitors program execution using `ptrace` and automatically packages up the **C**ode, **D**ata, and **E**nvironment required to run a set of x86-Linux programs on other x86-Linux machines [1].

The main benefits of CDE are that creating a package is completely automatic, and that running programs within a package requires no installation, configuration, or root permissions, thereby eliminating dependency hell.

The main limitation of CDE is that it is not guaranteed to find all the dependencies required for a complete package, so it is up to the user to insert additional files if necessary. Also, packages are only portable across machines with a compatible architecture and Linux kernel version. Despite these limitations, CDE has been downloaded over 2,000 times, and we have received hundreds of emails from users who have used it to quickly test and deploy software without installing any dependencies.

2 CDE system overview

We will use an example to introduce the core features of CDE. Suppose that Alice is a climate scientist whose experiment involves running a Python weather simulation script on a Tokyo dataset using this Linux command:

```
python weather_sim.py tokyo.dat
```

Alice's script (`weather_sim.py`) imports some 3rd-party Python extension modules, which consist of optimized C++ numerical analysis code compiled into shared libraries. If Alice wants her colleague Bob to run and build upon her experiment, then it is not sufficient to just send her script and `tokyo.dat` data file to him. Even if Bob has a compatible version of Python on his machine, he will not be able to run her script until he compiles, installs, and configures the extension modules that she used (and all of their transitive dependencies).

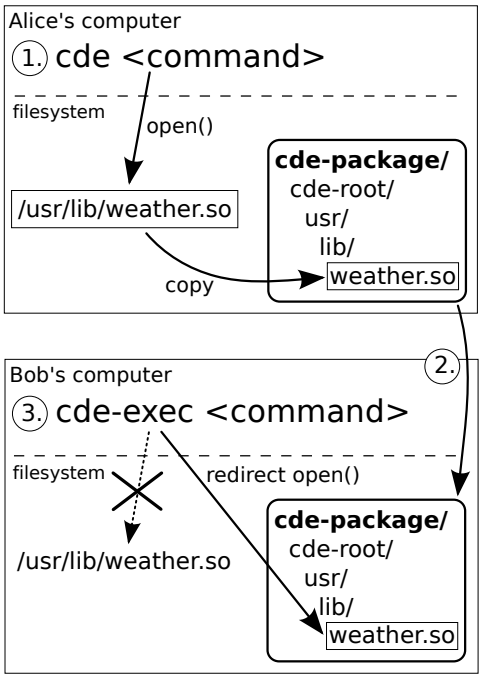


Figure 1: Example use of CDE: 1.) Alice runs her command with `cde` to create a package, 2.) Alice sends package to Bob's computer, 3.) Bob runs command with `cde-exec`, which redirects file accesses into package.

2.1 Creating a new package with `cde`

To create a self-contained package with all dependencies required to run her experiment on another machine, Alice prepends her command with the `cde` executable:

```
cde python weather_sim.py tokyo.dat
```

`cde` runs her command normally and uses the Linux `ptrace` mechanism to monitor all files it accesses throughout execution. `cde` creates a new sub-directory called `cde-package/cde-root/` and copies all of those accessed files into there, mirroring the original directory structure. For example, if her script dynamically loads an extension module (shared library) named `/usr/lib/weather.so`, then `cde` will copy it to `cde-package/cde-root/usr/lib/weather.so` (see Figure 1). When execution terminates, the `cde-package/` sub-directory (which we call a 'CDE package') contains all of the files and environment variables required to run Alice's original command.

2.2 Executing a package with `cde-exec`

Alice zips up the `cde-package/` directory and transfers it to Bob's Linux machine. Now Bob can run Alice's experiment without installing anything on his machine. He unzips the package, changes into the sub-directory containing the script, and prepends the original command with the `cde-exec` executable (also in the package):

```
cde-exec python weather_sim.py tokyo.dat
```

`cde-exec` sets up the environment variables saved from Alice's machine and executes the version of `python` and its extension modules from within the package. `cde-exec` uses `ptrace` to monitor all system calls that access files and rewrites their path arguments to the corresponding paths within the `cde-package/cde-root/` sub-directory. For example, when her script requests to load the `/usr/lib/weather.so` extension library using an `open` system call, `cde-exec` rewrites the path argument of the `open` call to `cde-package/cde-root/usr/lib/weather.so` (see Figure 1). This path redirection is essential, because `/usr/lib/weather.so` probably does not exist on Bob's machine.

Not only can Bob reproduce Alice's exact experiment, but he can also edit her script and dataset and then re-run to explore variations and alternative hypotheses, as long as he does not cause the script to import new Python extension modules that are not in the package.

3 Implementation

CDE uses the Linux `ptrace` system call to monitor the target program's processes, read and write to its memory, and modify its system call arguments, all without requiring root permission. We implemented CDE by adding 2500 lines of C code to the `strace` system call monitoring tool. The same ideas could be used to implement CDE for other architectures or operating systems.

3.1 Creating a new package with `cde`

Primary action: The main job of `cde` is to use `ptrace` to monitor the target program's system calls and copy all of its accessed files into a self-contained package. After the kernel finishes executing a syscall that takes a file path string as an argument (the 'File path access' category in Table 1) and is about to return to the target program, `cde` wakes and observes the return value. If the return value signifies that the indicated file exists, then `cde` copies that file into the package.

Prior to copying a file into the package, `cde` creates all necessary sub-directories and symbolic links to exactly mirror that file's location. If a file is a symlink, then both it and its target must be copied into the package.

If the copied file is an ELF binary, then `cde` searches its contents for constant strings that are filenames and then recursively copies those files into the package. This simple hack works well in practice to partially overcome CDE's limitation of only being able to gather dependencies on executed paths, since many binaries dynamically load libraries named by constant strings.

Category	Linux syscalls	cde action	cde-exec action
File path access	open[at], mknod[at], fstatat64 access, faccessat, readlink[at] truncate[64], stat[64], creat lstat[64], oldstat, oldlstat chown[32], lchown[32] fchownat, chmod, fchmodat utime, utimes, futimesat	Copy file into package	Redirect path into package
Local IPC sockets	bind, connect	none	Redirect path into package
Mutate filesystem	link[at], symlink[at] rename[at], unlink[at] mkdir[at], rmdir	Repeat in package	Redirect path into package
Get current dir.	getcwd	Update current dir.	Spoof current dir.
Change directory	chdir, fchdir	Update current working directory	
Spawn child	fork, vfork, clone	Track child process or thread	
Execute program	execve	Copy bin & linker into pkg	Maybe run dynamic linker

Table 1: The 48 Linux system calls intercepted by cde and cde-exec, and actions taken for each category of syscalls. Syscalls with suffixes in [brackets] include variants with/without the suffix: e.g., open[at] means open and openat.

Mutate filesystem: After each call that mutates the filesystem, cde repeats the same action on the corresponding copies of files in the package. For example, if a program renames a file from foo to bar, then cde also renames the copy of foo in the package to bar.

Updating current working directory: At the completion of getcwd, chdir, and fchdir, cde updates its record of the monitored process's current working directory, which is necessary for resolving relative paths.

Tracking sub-processes and threads: If the target program spawns sub-processes, cde also attaches onto those children with ptrace (it attaches onto spawned threads in the same way). cde keeps track of each monitored process's current working directory and shared memory segment address (needed for §3.2). cde remains single-threaded and responds to events queued by ptrace.

3.2 Executing a package with cde-exec

Primary action: The main job of cde-exec is to use ptrace to redirect file paths that the target program requests into the package. Before the kernel executes most syscalls listed in Table 1, cde-exec rewrites their path argument(s) to refer to the corresponding path within cde-package/cde-root/. By doing so, cde-exec creates a chroot-like sandbox that fools the program into 'believing' that it is executing on the original machine.

To reliably rewrite syscall arguments, cde-exec redirects the pointer to the argument's buffer. When a target process first makes a syscall, cde-exec forces it to make another syscall to attach a 16KB shared memory segment (a trick from [16]). Prior to every file path access syscall, cde-exec computes and writes the redirected path into shared memory and uses ptrace to mutate the syscall's argument, stored in a register, to point to the start of the shared memory segment in the target's address space.

Spoofing current working directory: At the completion of the getcwd syscall, cde-exec mutates (truncates) its return value string to eliminate all absolute path components up to and including cde-root/.

execve: When the target program executes a dynamically-linked binary, cde-exec rewrites the execve syscall arguments to execute the dynamic linker stored in the package rather than directly executing the binary. The dynamic linker on one distro might not be compatible with binaries created on another distro due to minor differences in ELF binary formats. Therefore, to maximize portability across machines, cde copies the dynamic linker into the package, and cde-exec executes the dynamic linker from the package rather than having Linux execute the system's version. Without this hack, even a trivial "hello world" binary compiled on one distro (e.g., Ubuntu with Linux 2.6.35) will not run on an older one (e.g., Knoppix with Linux 2.6.17).

Ignoring files and environment vars: By convention, Linux directories like /dev, /proc, and /sys contain pseudo-files that do not make sense to include in a CDE package. To improve package portability, we have manually created a user-customizable blacklist of a dozen directories, files, and environment variables for CDE to ig-

nore. `cde` will not copy ignored files (or vars) into a package, and `cde-exec` will not redirect their paths and instead access the real versions on the target machine.

4 Limitations

Executing a command within a CDE package will fail if:

- the arguments or input change to make the program load a new file (e.g., library, config file) that the original execution did not load. In general, *no automatic tool* (static or dynamic) can find all the dependencies required to execute all possible program paths, since that problem is undecidable. However, since a CDE package is just an ordinary directory tree, it is easy for users to directly add more files into the package if necessary. Also, if the user runs multiple commands in the same directory, `cde` will add additional files into the same `cde-package/`.
- the Linux kernel or hardware architecture on the target machine is incompatible with the binaries in the package. Mainstream distros contain libraries that are forwards- and backwards-compatible over several years. For example, the standard libs on 2010-era Ubuntu work on distros from as old as 2006 ($\geq$ 2.6.15 kernel), and the libs on 2007-era Fedora work on 2004-era distros ($\geq$ 2.6.9). Also, our intuition is that packages created today will run fine on Linux 2.6 distros from several years in the future, since kernel developers place high priority on maintaining backwards compatibility in the kernel-to-user ABI. Users who desire greater portability or 'future-proofing' can embed CDE packages within virtual machine or processor emulator images.

In addition, CDE is limited by the limitations of `ptrace` and of executing binaries by explicitly invoking the dynamic linker. `ptrace` can cause subtle differences in the semantics of traced processes, most notably that a process being monitored by `ptrace` cannot itself `ptrace` another process, which precludes the use of CDE alongside applications like symbolic debuggers. Also, there is a known bug on certain Ubuntu distros where the `bash` shell non-deterministically crashes when invoked explicitly with a dynamic linker; a workaround is to have CDE use the machine's native `bash` shell on those distros.

5 Real-world use cases

Since we released the first version of the CDE executable online on Nov 9, 2010, it has been downloaded at least 2,000 times (as of April 2011) [1]. We have exchanged hundreds of emails with CDE users and discovered six salient use cases as a result of our discussions. For our experiments (see Table 2), we used representative packages from each use case category (names in **`bold`**).

Distributing research software: The creators of two research tools — the **`arachni`** web app. security scanner [5] and the **`graph-tool`** math library [6] — used CDE to create portable binary packages that they uploaded to their project websites, so that their users do not have to go through the anguish of compiling them from source.

In addition, we used CDE to create portable binary packages for two of our Stanford colleagues' research tools, which were originally distributed as hard-to-compile source code tarballs: **`pads`** [11] and **`saturn`** [8].

Running software on incompatible distros: Even production-quality software might be hard to install on Linux distros with older kernel or library versions. For example, a Cisco engineer wanted to run some new open-source tools on his work machines, but the IT department mandated that those machines run an older, more secure enterprise Linux distro. He could not install the tools on those machines because that older distro did not have up-to-date libraries, and he was not allowed to upgrade. Therefore, he installed a modern distro at home, ran CDE on there to create packages for the tools he wanted to port (e.g., the **`meld`** visual text diff tool), and then ran the tools from within the packages on his work machines.

Hobbyists applied CDE in a similar way: A game enthusiast could only run a classic game (**`bio-menace`**) within a DOS emulator on one of his Linux machines, so he used CDE to create a package and can now play the game on his other machines. We also helped a user create a portable package for the Google Earth 3D map application (**`google-earth`**), so he can now run it on older distros whose libraries are incompatible with Google Earth.

Reproducible computational experiments: A fundamental tenet of science is that colleagues should be able to reproduce the results of one's experiments. Recently, some science journals and CS conferences are starting to encourage authors of published papers to put their code and datasets online, so that others can independently re-run, verify, and build upon their experiments. However, it can be hard to set up all of the (often-undocumented) dependencies required to re-run experiments.

Scientists can run the experiment once on their machine with CDE to create a package, and colleagues can run that package on any contemporary Linux machine to repeat the experiment. A robotics researcher used CDE to make the experiments for his motion planning paper (**`kpiece`**) [17] fully-reproducible. Similarly, we helped a social networking researcher create a reproducible package for his genetic algorithm paper (**`gadm`**) [15].

Deploying computations to cluster or cloud: Our colleague Peter wanted to use a department-administered

100-CPU cluster to run a parallel image processing job on topological maps (**ztopo**). However, since he did not have root access on those older machines, it was nearly impossible for him to install all of the dependencies required to run his computation, especially the image processing libraries. Peter used CDE to create a package by running his job on a small dataset on his desktop, transferred the package and the complete dataset to the cluster, and then ran 100 instances of it in parallel there.

Similarly, we worked with lab-mates to use CDE to deploy the CPU-intensive **klee** [10] bug finding tool from the desktop to Amazon's EC2 cloud computing service without needing to compile Klee on the cloud machines.

Submitting executable bug reports: Bug reporting is a tedious manual process. Users submit reports by writing down the steps for reproduction, exact versions of executables and dependent libraries, and maybe attaching an input that triggers the bug. Developers often have trouble reproducing bugs based on these hand-written descriptions and end up closing reports as "not reproducible."

CDE offers an easier solution: The reporter can simply run the command that triggers the bug under CDE supervision to create a CDE package, send that package to the developer, and the developer can re-run that same command on their machine to reproduce the bug. Three bug reporters sent us CDE packages, and we were able to reproduce all of their bugs: one that causes the Coq proof assistant to produce incorrect output (**coq-bug**) [2], one that segfaults the GCC compiler (**gcc-bug**) [3], and one that makes the LLVM compiler allocate an enormous amount of memory and crash (**llvm-bug**) [4].

Collaborating on class projects: Rahul, a Stanford grad student, was using the NLTK natural language processing module to build a semantic email search engine (**email-search**) for a machine learning class. Despite much struggle, Rahul's two teammates were unable to install NLTK on their machines due to conflicting library versions and dependency hell, so they only had one runnable copy. Rahul used CDE to create a package for their project and was able to run it on his two teammates' machines, so that all three of them could test and debug in parallel. Similarly, an undergrad used CDE to collaborate on and demo his virtual reality project (**vr-osg**).

6 Summary of experimental results

Due to space constraints, we summarize our main experimental results. Full details are in our tech report [12].

Package portability: To demonstrate that CDE packages can successfully execute on a range of Linux variants, we tested our benchmark packages on six popular distros, listed with the versions and release dates of their kernels:

Package name	Origin	Num libs	Slowdown
Distribute research software			
`arachni` [5]	2.6.35	48 (6)	
`graph-tool` [6]	2.6.26	149 (9)	
`pads` [11]	2.6.24⋆	9 (5)	28%
`saturn` [8]	2.6.18⋆	16 (8)	18%
Run production software on incompatible distros			
`meld`	2.6.35	93 (8)	
`bio-menace`	2.6.33	27 (26)	
`google-earth`	2.6.24 ⋆	82 (3)	19%
Create reproducible computational experiments			
`kpiece` [17]	2.6.35	30 (30)	
`gadm` [15]	2.6.18 ⋆	18 (4)	5%
Deploy computations to cluster or cloud			
`ztopo`	2.6.35	59 (35)	
`klee` [10]	2.6.32⋆	6 (6)	2%
Submit executable bug reports			
`coq-bug` [2]	2.6.32	3 (3)	
`gcc-bug` [3]	2.6.36	13 (2)	
`llvm-bug` [4]	2.6.35	8 (8)	
Collaborate on class programming projects			
`email-search`	2.6.32	138 (28)	
`vr-osg`	2.6.35	39 (28)	

Table 2: CDE packages by category. The 'Origin' column shows the kernel version where a package was created, and a star⋆ means it was created by the first author. The 'Num libs' column shows number of shared libraries (and number of statically-discoverable libs in parens).

1. CentOS 5.5 (Linux 2.6.18, Sep 2006)
2. Fedora Core 8 (Linux 2.6.23, Oct 2007)
3. openSUSE 11.1 (Linux 2.6.27, Oct 2008)
4. Ubuntu 9.10 (Linux 2.6.31, Sep 2009)
5. Mandriva Free Spring (Linux 2.6.33, Feb 2010)
6. Linux Mint 10 (Linux 2.6.35, Aug 2010)

Out of the 108 configurations we tested (18 CDE packages[1] each run on 6 Linux distros), *all executions succeeded* except for one (`vr-osg` failed on Fedora Core 8 with a known graphics-related error). By 'succeeded' we mean that the programs appeared to run correctly: Batch programs generated identical outputs across distros, and we could interact normally with GUI programs.

Necessity of dynamic tracking: We compared CDE against a static analysis that recursively runs the Linux `ldd` and `strings` utilities on executables files and libraries to find all string constants representing dependent

[1] Two of our benchmarks had both 32-bit and 64-bit versions.

libraries. Although this technique is simple, it represents what people actually do in practice, since it automates the tedious manual process of "chasing down and copying over dependent libraries" that folk wisdom suggests as the way to transport programs across machines.

The 'Num libs' column in Table 2 shows that in all but four benchmarks, the static technique found fewer libraries than CDE (the number of statically-discoverable libraries shown in parentheses). Thus, it cannot be used to create a portable package since the program will fail if *even one library is missing*. For similar reasons, static linking when compiling will not work either. This is why CDE's static+dynamic dependency tracking is necessary.

Run-time slowdown: We informally evaluated slowdowns on the five CDE packages we created (those marked with * in Table 2). Executing those programs within CDE packages were $2\% - 28\%$ slower than executing natively. The more system calls a program issues per second, the more CDE causes it to slow down, since the kernel must context switch to the CDE process during every syscall. We have heard that `ptrace` interposition can cause slowdowns of 10X or more, but we have not yet performed a rigorous performance stress test.

7 Related work

We know of no published system that automatically creates portable software packages *in situ* from a live running machine like CDE does. Existing tools for creating self-contained applications all require the user to manually specify dependencies. For example, Mac OS X programmers can create self-contained application bundles using Apple's developer tools. PDS is a prototype tool for creating self-contained Windows apps, which requires the user to manually specify a dependency list [9].

VMware ThinApp is a commercial tool that automatically creates self-contained portable Windows applications. However, a user can only create a package by having ThinApp monitor the installation of new software [7]. Unlike CDE, ThinApp cannot be used to create packages from existing software already installed on a live machine, which is our most common use case.

Virtual machine snapshots achieve CDE's main goal of capturing all dependencies required to execute a set of programs on another machine. However, they require the user to always be working within a VM from the start of a project (or else re-install all of their software within a new VM). Also, VM snapshot disk images are (by definition) larger than the corresponding CDE packages since they must also contain the OS kernel and other extraneous applications. CDE is a more lightweight solution because it enables users to create and run packages natively on their own machines rather than through a VM.

Finally, system call interposition using `ptrace` is a well-known technique that has been used for implementing tools such as secure sandboxes [13], record-replay systems [14], and user-level filesystems [16].

Acknowledgments: Thanks to Fernando Perez for the serendipitous discussion of reproducible research that planted the seeds of the idea for CDE, to Richard Spillane for sharing his Goanna code [16], to Imran Haque for the Slashdot publicity, to our users for their bug reports and feedback, and to {`ridddlr`, `paboonst`, `cbird`, `TomZ`, `ewencp`, `ihaque`, `daramos`} for editorial help. This research was supported by the NSF Graduate Research Fellowship and the United States Air Force Research Laboratory (AFRL) through Contract FA8650-10-C-7024.

References

[1] CDE project home page, `http://www.pgbovine.net/cde.html`.

[2] Coq proof assistant: Bug 2443, `http://coq.inria.fr/bugs/show_bug.cgi?id=2443`.

[3] GCC compiler: Bug 46651, `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=46651`.

[4] LLVM compiler: Bug 8679, `http://llvm.org/bugs/show_bug.cgi?id=8679`.

[5] `arachni` project home page, `https://github.com/Zapotek/arachni`.

[6] `graph-tool` project home page, `http://projects.skewed.de/graph-tool/`.

[7] VMware ThinApp User's Guide, `http://www.vmware.com/pdf/thinapp46_manual.pdf`.

[8] AIKEN, A., BUGRARA, S., DILLIG, I., DILLIG, T., HACKETT, B., AND HAWKINS, P. An overview of the Saturn project. PASTE '07, ACM, pp. 43–48.

[9] ALPERN, B., AUERBACH, J., BALA, V., FRAUENHOFER, T., MUMMERT, T., AND PIGOTT, M. PDS: a virtual execution environment for software deployment. VEE '05, ACM, pp. 175–185.

[10] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI '08, USENIX Association, pp. 209–224.

[11] FISHER, K., AND GRUBER, R. PADS: a domain-specific language for processing ad hoc data. PLDI '05, ACM, pp. 295–304.

[12] GUO, P. J., AND ENGLER, D. CDE: Using system call interposition to automatically create portable software packages. Stanford University Computer Science Technical Report 2011-01.

[13] JAIN, K., AND SEKAR, R. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. NDSS '00.

[14] LAADAN, O., VIENNOT, N., AND NIEH, J. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. SIGMETRICS '10, pp. 155–166.

[15] LAHIRI, M., AND CEBRIAN, M. The genetic algorithm as a general diffusion model for social networks. In *Proc. of the 24th AAAI Conference on Artificial Intelligence* (2010), AAAI Press.

[16] SPILLANE, R. P., WRIGHT, C. P., SIVATHANU, G., AND ZADOK, E. Rapid file system development using ptrace. In *Experimental Computer Science* (2007), USENIX Association.

[17] SUCAN, I. A., AND KAVRAKI, L. E. Kinodynamic motion planning by interior-exterior cell exploration. In *Int'l Workshop on the Algorithmic Foundations of Robotics* (2008), pp. 449–464.

Vsys: A programmable sudo

Sapan Bhatia*, Giovanni Di Stasi†, Thom Haddow‡, Andy Bavier*, Steve Muir♯, Larry Peterson*
*Princeton University †University of Napoli ‡Imperial College ♯Juniper Networks

We present Vsys, a mechanism for restricting access to privileged operations, much like the popular `sudo` tool on UNIX. Unlike `sudo`, Vsys allows privileges to be constrained using general-purpose programming languages and facilitates composing multiple system services into powerful abstractions for isolation. In use for over three years on PlanetLab, Vsys has enabled over 100 researchers to create private overlay networks, user-level file systems, virtual switches, and TCP-variants that function safely and without interference. Vsys has also been used by applications such as whole-system monitoring in a VM. We describe the design of Vsys and discuss our experiences and lessons learned.

1 Introduction

One of the key challenges we have faced when operating PlanetLab [1, 11] is helping researchers to implement and evaluate new ideas while maintaining a reasonable level of isolation between experiments (each of which runs in a separate *slice*). PlanetLab users may require the ability to sniff a subset of network traffic for diagnostic purposes, gain access to certain log data restricted to administrators, view global system state that is typically hidden from users, reserve TCP and UDP ports, create IP-level rules, and so on. We have received these requests frequently and continue to do so today [13]. Our goal is to grant such privileges to enable research, while simultaneously preserving isolation and the principle of least privilege to the extent possible.

Service isolation can be imposed at multiple levels in any system. A current trend is to equate virtual machines with service isolation, but different degrees of isolation can be enforced by the hardware, virtual machines, operating system, and user-space tools. On a PlanetLab node, Linux-Vservers [15] run each experiment in a `chroot` environment that prevents cross-domain actions between slices, and that provides a "superuser" account with limited privileges. However, since all slices share a single OS kernel, it is possible to grant additional OS privileges to a particular slice, for example, to let the "superuser" bind privileged ports or add routes to the kernel's IP forwarding table. But it is these sorts of these privileges that, if misused, can unacceptably impact other slices. We would like to limit a particular user to *only* bind port 53 to run his DNS service, and to *only* change routes on virtual devices that he controls. The problem is that the abstractions the OS gives us do not support granting privileges to users while imposing narrow limits on how they are used.

In this paper we describe Vsys, a framework that allows users to invoke privileged operations via scripts called *extensions* that precisely specify how these operations can be accessed. Vsys is inspired by the UNIX philosophy of creating new system services by combining simple OS primitives: Vsys enforces security policies and achieves isolation through a combination of existing OS primitives. For example, packet filters can block a subset of IP traffic from a service, virtual devices combined with bridging can be used to filter Ethernet traffic, `grep` can filter access to files, IP policy routing can instantiate key-based routes for packets, and so on.

Vsys began with the modest goal of being a `sudo` compatible with `chroot` jails. The `sudo` [17] tool allows users to run programs with the privileges of another user. It enables coarse-grained admission control via an access control list of commands that each user is allowed to run, along with limited predicates on the arguments. Vsys is designed with three primary goals that make it an improvement over `sudo`: (1) ease of assembling new extensions from existing OS abstractions and tools, using arbitrary programming languages; (2) ease of accessing extensions from the UNIX command line or within arbitrary programs; and (3) maintaining a fine-grained level of control over exactly what extensions users are able to invoke and how. With Vsys, simple tools can be used to rapidly develop extensions that multiple services can access safely. Unlike modifications to the virtual machine or OS layers of the system, a new Vsys extension can be developed and deployed on PlanetLab in a matter of days and enhanced incrementally over time.

This paper makes three contributions. First, we discuss the design of Vsys, an important feature of which is an Access Control Policy (ACP). Vsys ACPs insert policy code between the user, the Vsys extension, and the OS to ensure that the invocation of a given building-block command is consistent with the privilege granted to the user. Second, with the help of four heavily-used Vsys extensions, we show that existing OS primitives can be composed into powerful isolation abstractions, enabling functions such as virtual networking. While variants of the security mechanisms underlying Vsys have been explored before, Vsys is novel both in its details, and in its scope—for instance, in how extensions crosscut multiple OS subsystems (packet filtering, rout-

ing, sockets, file systems, etc.). Furthermore, Vsys has been used on a large scale for several years. Finally, we describe our experiences with Vsys and draw some lessons on creating new abstractions and fostering an active user community. We hope that these lessons and experiences will be helpful to designers of systems services and frameworks.

2 User View

In this section we describe Vsys from the standpoint of how extensions are added and invoked. In the next section we explain how Vsys works.

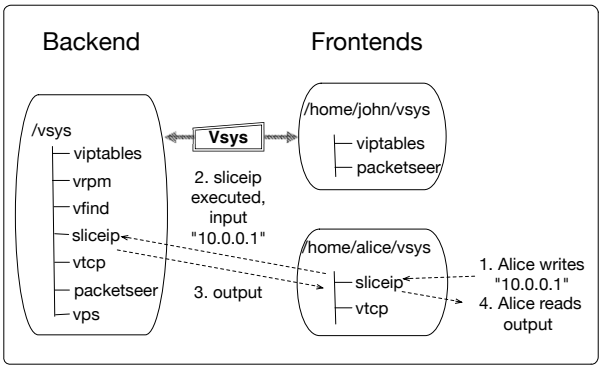


Figure 1: Basic use of Vsys

Figure 1 illustrates the basic operation of Vsys. Vsys extensions are executable scripts placed in a *backend* directory. Vsys monitors the backend directory and detects when new extensions are copied in. For each extension in the backend directory, Vsys also expects a corresponding ACL of the *contexts* authorized to use the extension. A context is just a system identifier like a slice or user ID. Vsys then creates special files (a pair of FIFO pipes or a UNIX domain socket) in the *frontend* directories for contexts listed on the extension's ACL. Each pair of FIFOs (for extensions with text input and output) or UNIX domain socket (for passing arbitrary data types) in a frontend directory maps to a specific extension in the backend directory.

Vsys requires that the special files it creates can only be read and written by contexts authorized to access the Vsys extension. In the `chroot` environment used on PlanetLab, each frontend directory is only visible within one slice's filesystem and so access to the FIFOs or socket is limited to the slice's users. On a standard UNIX system, the frontend could be any directory (e.g., a subdirectory of the user's HOME directory), and file system permissions limit access to a specific user.

In order to use an extension, a user simply opens the FIFO pipes or connects to the UNIX socket bearing the name of the extension and writes some arguments to it (e.g., to use a Vsys extension named `sliceip`, the user opens FIFOs called `/vsys/sliceip.in` and `/vsys/sliceip.out`). Vsys reads the arguments, runs the appropriate executable script on these arguments with sufficient privileges, and returns the output to the user through the pipe or socket.

3 Vsys Design

Vsys is intended to dispatch requests from non-privileged users to privileged extensions in a controlled manner. While there could be many approaches to implementing this functionality, we started with three design requirements. First, the Vsys framework should leverage existing UNIX primitives where possible. The philosophy of reusing OS building blocks when creating services inspired us to create Vsys; the Vsys design should also follow this philosophy. Second, users should be able to invoke Vsys using native operations on UNIX and on the command line, rather than via a new API or protocol. Third, one needed to be able to develop Vsys extensions using native code in any programming language. Our goal was to bundle Vsys as close to the OS as possible, not tying it with proprietary libraries, and to encourage users and administrators to contribute extensions by letting them program in their preferred programming environment.

These requirements led us to model our interface after the *everything-is-a-file* idiom as in Plan9 [12]. Users see Vsys extensions as special files in a */vsys* directory, and the Vsys daemon dispatches events back and forth between these special files and processes running extensions. The files that users interact with can be FIFO pipes or UNIX domain sockets. While the former are convenient to use, the latter support sending and receiving objects such as file and socket descriptors.

Vsys extensions are associated with access control policies (ACPs). An ACP is a program that defines a filter on the arguments passed to an operation, admitting a caller into the guarded operation only if the combination of the arguments and the current calling context is allowed by its policy. Each privileged operation wrapped by a Vsys extension is associated with two ACPs: an *invocation ACP* and a *syscall ACP*. The invocation ACP is run before the Vsys extension is executed and filters the arguments passed to the extension. The syscall ACP is triggered every time the extension makes a system call.

Figure 2 provides an overview of how Vsys works. Referring to the circled numbers:

1. A client process writes arguments into the input FIFO or UNIX domain socket corresponding to a particular Vsys extension. Vsys leverages UNIX file permissions and chroot to limit access to the FIFO or socket to a particular context (e.g., to a UID or a PlanetLab slice).

2. The Vsys daemon reads the arguments from the input FIFO or socket and looks up the corresponding

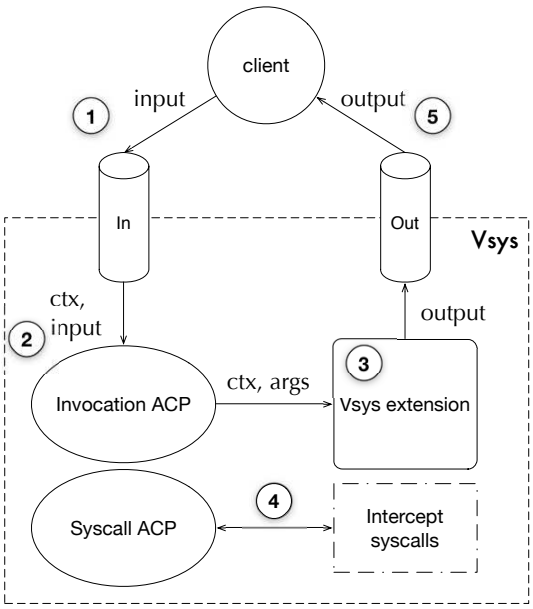


Figure 2: Vsys control flow

context. It passes the context ID and the arguments to the extension's invocation ACP. The invocation ACP runs and returns either success or failure.

3. If the invocation ACP returns success, the Vsys daemon executes the appropriate extension, passing in the original arguments and the calling context.

4. Vsys uses UNIX's `ptrace` facility to intercept system calls made by the extension. For each system call, Vsys executes the corresponding syscall ACP to allow or deny the call. This enables Vsys to limit the extension to touching specific resources (e.g., only opening certain ports).

5. Vsys writes output from the extension to the output FIFO or UNIX domain socket, from which it can be read by the client process.

Just like Vsys extensions themselves, ACPs are executables written in arbitrary programming languages. Sticking to our belief in re-using existing tools, we did not invent a new language to implement ACPs. In most cases, policies can be encoded with the help of regular expressions. Alternatively, lexing and parsing libraries such as GNU lex and yacc may also be used.

The design of Vsys relies on our assumed threat model: that extension developers are trusted, even though end users are not. On PlanetLab, all Vsys extensions are vetted by an administrator before they are deployed. We believe that the Vsys design provides a reasonable amount of security against the threat of malicious users. The Vsys daemon is simple and does little more than read and write FIFOs and execute processes. It is written in Ocaml [21], a type-safe functional language that adds robustness to this simplicity. The boundary between end users and Vsys extensions is stringent and cannot be circumvented as Vsys extensions run in a separate process address space. Extensions themselves are controlled using `ptrace`, which is a weak security mechanism [5]. However, security at that layer focuses on narrowing the interface to system calls that may be called with tainted data—i.e., inputs from an end user. It does not protect against malicious extension developers. Finally, we expect Vsys extensions to follow good coding practices by checking user-provided data and composing provably correct inputs to sensitive operations, as opposed to passing such tainted data directly or a transformed version of it.

4 Vsys Extension Library

An active user community has contributed a number of powerful extensions to Vsys over the years. This section presents several extensions that have been deployed and used on PlanetLab.

4.1 sliceip

`sliceip` enables users to create service-specific route tables. It is invoked with the same syntax as the `ip` command that creates and manages routes on Linux.

`sliceip` implements isolation through IP policy routing, a mechanism that extends the definition of the hash key used in routing to include fields other than the destination IP address. `sliceip` uses a *packet tag*, which associates packets with the sending or receiving user, as part of this route key. Thus, even when two users define separate routes to a single destination address, the tag determines whether a packet should take the route defined by the first user, the second user, or whether it should take the default route. There are many ways to set this packet tag to associate packets with users. The easiest way is to use the intermediate step of a network interface. The `pl_tuntap` extension discussed below lets users create and manage isolated virtual interfaces. Since local packets hold a record of the interface that emitted them, the name or ID of this interface can be used as the packet tag to identify the user.

Combined with `pl_tuntap`, `sliceip` enables users to create virtual overlay networks—one problem that the Linux community tackled by implementing the `netns` module for the Linux kernel. In contrast to `netns` which took over four years to develop and is still under active development, the deployment timeline for `sliceip` was of the order of months.

4.2 fusemount

FUSE [10] is a Linux-based framework for implementing and managing filesystems in userspace. A new filesystem can be developed by implementing standard filesystem operations such as directory and file lookups,

and exporting these operations via the FUSE userspace library. An in-kernel component implemented as a kernel module and the FUSE library communicate via a file descriptor obtained by opening a special character device (`/dev/fuse`). The obtained file descriptor is subsequently passed to the mount system call, to match up the descriptor with the mounted filesystem.

FUSE facilitated the development and deployment of the WheelFS [16] wide-area distributed filesystem on PlanetLab. WheelFS is implemented as a FUSE module that can be instantiated by PlanetLab users via Vsys. The authors of this work have made it possible for PlanetLab users to create their own shared filesystem as well as share it with other users [20].

Unlike `sliceip`, the `fusemount` extension is accessed via a UNIX domain socket. The caller (i.e., the user creating the filesystem) first obtains a file descriptor and uses it to populate a virtual filesystem via FUSE. Since at this point the filesystem has not been mounted, the operation of obtaining and using the file descriptor is safe. Next, the user connects to `fusemount` by opening the corresponding UNIX domain socket, and passes the aforementioned file descriptor over this connection. `fusemount` then performs the mount operation via the received file descriptor and passes the file descriptor back to the caller. The ensures the restrictions of the mount operation, such as by making sure that the mount point is owned by the caller.

4.3 socketops

`socketops` is a collection of extensions that lets users create privileged sockets for operating large TCP or UDP buffers, viewing low-level packet headers, etc. Rather than granting coarse-grained administrative access to the network as facilitated by the `CAP_NET_ADMIN` capability on Linux, Vsys allows users to access these operations selectively. Similar to `fusemount`, all of the above operations are accessed via UNIX domain sockets. Callers open the domain socket corresponding to the desired extension and pass parameters, such as a buffer size for bmsocket. The Vsys extension then returns a socket descriptor with the requested properties, and can be used by the caller independent of Vsys.

4.4 vtuntap

`vtuntap` lets users create and manage virtual devices without giving them administrative access to the network. This extension is a wrapper around *tun/tap*, a popular virtual point-to-point network device on Linux. On Linux, the tun/tap device is used via a file descriptor obtained by opening a special character device (`/dev/tun`). The file opening operation causes the tun/tap kernel module to create a new network interface. The device can then be configured using tools such as *ifconfig*. Once configured, the kernel serializes all packets sent to the device as a raw stream of packet data to the aforementioned file descriptor, and receives data written to the file descriptor as packets on the device.

`vtuntap` arbitrates both steps in this operation via two extensions. The `pl_tuntap` extension uses the UNIX socket interface to create the tun/tap file descriptor and send it to the caller. In addition, the `vif_up` extension lets users configure devices with parameters such as the MTU, transmit queue length and the IP address. `vif_up` is a wrapper around the `ifconfig` command and takes the same set of parameters. Through an ACP, Vsys verifies that the user is restricted to a set of allowed IP addresses and other authorized parameters.

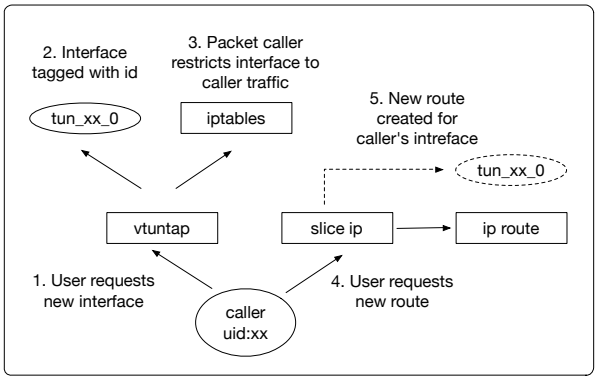


Figure 3: The vtuntap extension

Figure 3 shows how a user can combine `vtuntap` with `sliceip` to create an overlay topology. The user first calls `vtuntap` to create a new network interface, tag it an ID field unique to the user, and restrict the traffic to that interface. This configures the overlay's data plane. Then `sliceip` can add and remove routes on the control plane of the overlay.

5 Experiences and Lessons

This section articulates some of our experiences building Vsys and the conclusions to which they have led us.

Creating new OS abstractions is hard. The goal of the VINI project was to build a new testbed, using PlanetLab software, that would combine isolated virtual network topologies and PlanetLab slices [2]. At the start of the project, we considered two alternatives. The first was building virtual topologies using existing Linux networking primitives (e.g., IP policy routing) and user-space tools. We initially developed Vsys to explore this space. The second alternative was to leverage Linux network namespaces (`netns`), a new abstraction for the Linux kernel intended specifically for isolating the network subsystem. Over a summer we developed an initial prototype using `netns` that satisfied the requirements of

VINI and deployed it on the public VINI testbed [3].

Our subsequent experiences with `netns` were less positive. On the VINI testbed, incremental improvement of our `netns`-based prototype as well as bug fixes to `netns` continued for about two years. By this point the Vsys-based approach for building virtual topologies, which we had deployed on PlanetLab, was mature and had been used extensively by researchers. Most of the issues we encountered on VINI involved interactions between `netns` and other modules. For example, PlanetLab and VINI use Linux-Vserver to assign IP addresses to network slices, but `netns` would hide network devices from Vservers. The tools associated with `netns` unexpectedly also added filesystem and namespace isolation to processes when only network isolation was requested. Upon modifying the tools, we realized that there were dependencies between these isolations that required further kernel modifications. Such problems were hard to diagnose because of the lack of debugging tools for the new abstractions.

The lesson is that the continued predominance of old abstractions (e.g., pipes, file descriptors, and sockets) is no coincidence. Since fundamental OS abstractions are global and can affect all modules and processes in the system, changes cause side effects that are very hard to predict, especially when these side effects cut across modules. Had we foreseen our problems with `netns`, we would probably have focused our efforts on the Vsys approach from the start.

Flexibility drives innovation in development. Though we invite the PlanetLab user community to contribute code, we receive few contributions. Vsys extensions have been the exception to this rule: all but one Vsys extension were submitted by developers other than the authors of the Vsys framework. Vsys is a success story in our efforts to engage PlanetLab users in helping to develop the platform. This success is all the more surprising given that such user contributions are unusual for security mechanisms.

We attribute this to the use of standard abstractions and the ability to use the programming language of one's choice. In Vsys, an extension is an executable script to which inputs are passed explicitly as arguments and via standard input. This explicit and simple data flow adds developer confidence to the reliability of a script and enables him or her to develop scripts on a standard installation of Linux even if it does not run the PlanetLab environment. The ability to use any programming language also helps contributors reuse their existing code, regardless of the language it is written in.

The lesson is that even security mechanisms can attract external developers if you provide a flexible and easy-to-learn development environment. In our experience, skilled developers also have very strong tastes for using specific programming tools and standard environments, which it helps to support.

Reusing standard abstractions simplifies interfaces between components. Despite having been developed independently by over a dozen individuals, many of PlanetLab's Vsys scripts depend on one another. For instance, the `fd_tuntap` and `reserve_eth` scripts allocate network endpoints for users, and `vifconfig` configures the parameters of these devices and sets up routes and network address translations. Similarly, `sliceip` sets up tunnels and `makeswitch` connects the interfaces to virtual OpenFlow switches. The lesson is that the use of standard primitives—files, file descriptors, pipes, directories, network interfaces, packet filtering rules, network routes, and sockets—simplifies interfaces and facilitates program reuse.

6 Related Work

There is a great deal of work related to Vsys; we will focus on UNIX-centric mechanisms based on processes and other standard OS primitives. Vsys is similar to existing sandboxing tools [17, 6, 7, 5, 19, 4] but is novel in both details and scope. Furthermore, unlike those systems, Vsys has been deployed and used at scale for several years. In the process it has attracted numerous contributions from an active user community, validating our design goal of flexibility through the use of grassroots abstractions.

Vsys is similar to the interposition agents introduced by Jones [9] to insert policy between privileged operations and untrusted user code. Jones implemented a library of object-oriented abstractions that could be used to intercept system calls and modify their behavior, such as by tracing them or filtering their arguments. Vsys divides policy code between extensions and ACPs. Thus rather than one, there are two interposition sites, the first between calling clients and the underlying OS (i.e. the extensions themselves), and the second between the extensions and the underlying OS (i.e. the syscall ACPs).

SLIC [6], Janus [7] and Ostia [5] are sandboxing frameworks that use system call filtering and delegation to grant untrusted processes access to system resources. In Vsys, isolation is implemented in the form of Vsys extensions, which compose multiple system abstractions such as file descriptors, sockets and packet filtering rules. Like Janus, Vsys uses system call filtering with the help of `ptrace` to reinforce the limitations that the extension developer places on clients. Ostia uses system call delegation to protect against time-of-check-to-time-of-use bugs. The delegation mechanism executes the system call on the client's behalf immediately after authenticating it, eliminating the window of opportunity for attackers. The Vsys design assumes that extension developers are not malicious and so such mechanisms

are unnecessary; the goal of system call filtering in Vsys is to narrow the interface to extensions, as a backup to incomplete checks by script developers. Jain and Sekar's framework [8] also uses system calls for containment. Finally, Systrace [14] enables administrators to define system call access policies in much the same way as UNIX permissions define file access policies. In this way, fine-grained control can be imposed on processes, and the privileges of programs can be elevated without the use of potentially dangerous suid binaries.

The Proper ("PRivileged OPERations") daemon was the precursor to Vsys. It let PlanetLab users run privileged operations by passing file descriptors between privileged and non-privileged contexts. Users invoked primitive operations—socket creation, file opening and closing, execution, etc.—proxied by the Proper daemon. Vsys inherits Proper's use of file-descriptor passing from privileged to non-privileged contexts.

A more advanced form of `sudo` is `sus` [18], which extends the access control list to include predicates on objects such as files and users. Calife [4] is another variant of `sudo` with usability enhancements and privileged command logging. `SSU` [19] handles the remote execution of privileged operations over `ssh` sessions.

In contrast to these tools and their variants, the goal of Vsys goes beyond defining ACLs for privileged commands. Vsys is meant to facilitate the composition of existing tools to build isolated operations. The relationship between `sudo` scripts and Vsys extensions can be compared to that between assembly language and high-level programming languages. The former is a low-level mechanism and the latter provides convenient abstractions such as ACPs, context identifiers and file descriptor transfers making use of the mechanism.

Perhaps the best known OS mechanism for privilege allocation is UNIX file permissions and `setuid` bits. Vsys sets permissions on pipes and sockets so that they can only be opened by users authorized to access the corresponding extensions. Vsys is also able to penetrate `chroot` jails and filesystem containers, letting users invoke functionality that they do not have direct access to in the filesystem.

7 Conclusion

It has been demonstrated many times over that the rich library of abstractions available on OSes can go a long way in solving problems for which dedicated OS extensions were developed. Our experiences with Vsys have reinforced this belief by showing that simple compositions of existing UNIX tools can be used to implement powerful isolation. The `sliceip` and `pl_tuntap` extensions enable network isolation comparable to that found in dedicated approaches such as VINI and Linux network namespaces. The `fuse` extension lets users create userspace filesystems in a collaborative manner, letting other users on the system mount and use deployed filesystems. Our design choice of grassroots abstractions and an unconstrained development environment has also been validated by the continuous contributions of Vsys extensions by an active user community.

References

[1] BAVIER, A., BOWMAN, M., CULLER, D., CHUN, B., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating System Support for Planetary-Scale Network Services. In *Proc. 1st NSDI* (San Francisco, CA, Mar 2004).

[2] BAVIER, A., FEAMSTER, N., HUANG, M., PETERSON, L., AND REXFORD, J. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. SIGCOMM 2006* (Pisa, Italy, Sep 2006).

[3] BHATIA, S., MOTIWALA, M., MUHLBAUER, W., MUNDADA, Y., VALANCIUS, V., BAVIER, A., FEAMSTER, N., PETERSON, L., AND REXFORD, J. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proc. ACM CoNEXT 2008* (Madrid, Spain, December 2008).

[4] Calife: how to become root (or another user) with ones own password. http://www.keltia.net/programs/calife/.

[5] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proc. 2004 Symposium on Network and Distributed System Security* (2004).

[6] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., AND ANDERSON, T. E. Slic: an extensibility system for commodity operating systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (1998), ATEC.

[7] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography - Volume 6* (1996).

[8] JAIN, K., AND SEKAR, R. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *In Proc. Network and Distributed Systems Security Symposium* (1999).

[9] JONES, M. B. Interposition agents: transparently interposing user code at the system interface. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (1993), SOSP.

[10] Filesystem In Userspace. http://fuse.sourceforge.net.

[11] PETERSON, L., BAVIER, A., FIUCZYNSKI, M., AND MUIR, S. Experiences Building PlanetLab. In *Proc. 7th OSDI* (Seattle, WA, Nov 2006).

[12] PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference* (1990), pp. 1–9.

[13] Questions on PlanetLab Development Mailing List. http://lists.planet-lab.org/pipermail/devel/2009-December/004014.html http://lists.planet-lab.org/pipermail/devel/2009-June/003470.html http://lists.planet-lab.org/pipermail/users/2010-November/003756.html.

[14] PROVOS, N. Improving host security with system call policies. In *In Proceedings of the 12th Usenix Security Symposium* (2002).

[15] SOLTESZ, S., POTZL, H., FIUCZYNSKI, M., BAVIER, A., AND PETERSON, L. Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors. In *Proc. EuroSys 2007* (Lisbon, Portugal, Mar 2007).

[16] STRIBLING, J., SOVRAN, Y., ZHANG, I., PRETZER, X., LI, J., KAASHOEK, M. F., AND MORRIS, R. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)* (April 2009).

[17] Man page for sudo. http://www.gratisoft.us/sudo/sudo.man.html.

[18] Sus privilege elevation tool. http://pdg.uow.edu.au/sus/.

[19] THORPE, C. Ssu: Extending ssh for secure root administration. In *Proceedings of the 12th USENIX conference on System administration* (1998).

[20] WheelFS Installation Instructions for PlanetLab. http://pdos.csail.mit.edu/wheelfs/doku.php?id=documentation#planetlab_nodes.

[21] XAVIER LEROY. The objective caml system, release 1.07, documentation and user's manual., 1997. http://caml.inria.fr/pub/distrib/ocaml-1.07/ocaml-1.07-refman.txt.

Internet-scale Visualization and Detection of Performance Events

Jeffrey Pang, Subhabrata Sen, Oliver Spatscheck, Shobha Venkataraman
AT&T Labs - Research, Florham Park, NJ, USA

1 Introduction

Network server farms host a wide range of important applications, such as e-commerce, content distribution, and cloud services. Because server farms serve customers spread across the Internet, the key to effective server farm management is the ability to detect and resolve problems between a farm and its clients. Operators typically monitor performance using rule-based scripts to automatically flag "events of interest" in an array of active and passive measurement feeds. While effective, these rule-based approaches are usually limited to events with known properties. Equally important to operators is finding the "unknown unknowns" — novel events of interest with properties that have not been observed before. Effective visualization greatly aids in the discovery of such events, as operators with domain expertise can quickly notice unexpected performance patterns when represented visually. This paper presents *BirdsEye*, a tool that visualizes performance at Internet scale.

Designing such a tool is non-trivial because operators have to diagnose performance problems that may manifest themselves anywhere on the Internet. Visualizing all the possible ways these problems may manifest themselves poses three challenges: First, the vastness of the Internet and the sheer volume of raw performance data make it impossible for a human operator to comprehend every piece of information about every part of the Internet. An effective visualization needs to be sparse in representation, yet discriminating of good and poor performance. Second, problems can manifest themselves at multiple scales – e.g., a degraded peering link might impact entire swaths of the IP address space while a misdirected client might only affect a single ISP. Thus, there is not a single "level" of monitoring that can capture all problems that operators care about. Finally, performance problems not only correlate across space, but also across time – e.g., a problem may occur periodically during a certain time of the day. Thus, an effective visualization must present both the spatial view of performance and show how it changes over time.

To meet these challenges, we first observe that a tree is a natural way to visualize the Internet performance from the perspective of a server farm. That is, the IP address hierarchy can be interpreted as a tree with each node corresponding to an IP prefix, and its children corresponding to sub-prefixes. If we color a node (e.g., progressively from green to red) based on the likelihood that an IP address in that node's prefix is experiencing a performance problem, we will likely be able to differentiate "good" portions of the address space vs. "bad" portions. This is because IP addresses in the same prefix are more likely to be geographically close, under the same administrative control, and/or share the same routing paths. Thus, their performance is likely to be correlated. The focus of this paper is how to visualize this tree effectively.

A straight-forward approach would be to visualize the entire tree up to a predetermined aggregation-level, such as BGP prefixes. But this approach either would not sparse enough for human comprehension, or would not represent problems at granularities other than the predefined one. Instead, BirdsEye builds *adaptive decision trees* over the IP address space using recent performance measurements. These decision trees group IP addresses with similar performance characteristics and separate those with significantly different performance characteristics. Moreover, these trees are learned online, and adapt to changes in the underlying network. Therefore, changes in performance are reflected in the decision tree over time. By visualizing these adaptive decision trees, BirdsEye shows the performance to the entire Internet, but only highlights the parts that have bad performance at any given point in time.

We present an evaluation of our tool using more then 50 million Round Trip Time (RTT) measurements collected from a distributed server farm in a tier-1 ISP. While RTTs are not the only performance measurements we can visualize, they are one important metric of interest tracked by many operators. Through this case study,

we discover several RTT anomalies, such as diurnal patterns of poor performance in particular access ISPs and an ISP that was likely misdirected by the server farm. This was unknown to operators, suggesting that BirdsEye is indeed useful in finding novel performance problems. We envision that BirdsEye will supplement existing rule based systems — once an operator has verified that a hitherto unknown pattern deserves more attention, they can create new rule-based scripts to flag the patterns.

2 Design Overview

2.1 Design Requirements

The challenges described in Sec. 1 dictate four requirements for visualizing the Internet tree:

Sparse Network-wide Representation. The visualization of the tree needs to encompass the entire Internet in order to able to pinpoint any region with performance problems. However, in order to be usable by human operators, the tree also needs to be sparse, highlighting only the regions needed to differentiate performance experienced by clients. We ensure a sparse representation by enforcing a limit on the maximum number of leaves the Internet tree can have.

Multi-Level Drill-Down. While being sparse, the tree should not overly focus on a single level of the address space such as /24s or /8s since problems may manifest themselves at multiple levels in the hierarchy. For example, our case studies show that there are scenarios where large prefix ranges (e.g., /10 blocks) can be combined because they all experience the same performance. However, there are other scenarios where small ranges (e.g., belonging to management systems) must be identified individually to ensure that their performance is monitored. Since the depth of each branch in the tree represents how far operators can visually drill-down into a given prefix, our tool automatically infers the depth needed to differentiate performance among IPs in each prefix.

Capture Temporal Dynamics. The tree also needs to reflect changes in the performance of clients across the Internet. For example, by looking at a time series of trees, an operator should be able to quickly see prefixes that deviate from normal performance, e.g., due to a degraded peering link. We address this requirement by computing an *adaptive* tree. That is, it can modify its structure and performance indicators over time as more measurements are received.

Real-time Rendering. Finally, the tree needs to be constructed efficiently over large volumes of performance measurements, and updated periodically (e.g. every 5 minutes) to allow operators view the performance measurements in a timely fashion. We address this requirement by ensuring that the tree can be constructed in an online fashion over a stream of performance data.

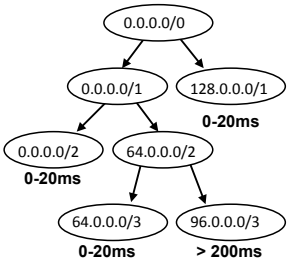


Figure 1: An example IPTree with 4 leaves.

2.2 Tool Overview

BirdsEye has two main components: (1) the *tree-constructor*, which generates Internet trees that meet the aforementioned design requirements, and (2) the *visualizer* which generates visualizations of the tree, in a manner that highlights anomalies and changes in performance. The tree-constructor takes as input a stream of performance data (e.g., RTT measurements). Each time interval, it sends an updated tree to the visualizer. The visualizer then generates and displays a graphic for the updated tree, using each node's performance indicators to colour it. We describe the construction of the tree in detail in Section 3, and visualization in Section 4. In Section 5, we show, using real-world examples, that the time series of Internet trees visually highlights both regular and irregular performance patterns.

3 Generating Internet Trees

In this section, we describe our algorithm for generating accurate Internet trees, focusing on latency measurements as a concrete example of performance data. Since the Internet tree needs to differentiate between client IP addresses based on their performance, it effectively builds a decision tree over the IP address hierarchy. Automatically inferring such an Internet tree from performance data is thus a decision-tree learning problem, quite different from hierarchical heavy-hitter problems (see Sec. 6). Note, however, that this decision tree is quite different from decision trees typically built in diagnosis applications – our tree is based purely on the structure of the IP address hierarchy. This, along with our requirements, make it infeasible to apply standard decision-tree learning algorithms. Instead, we extend the algorithmic framework proposed in [8] for latency prediction at server farms, since this framework incorporates our design requirements (i.e., learning over streaming data, sparse representation, fundamentally adaptive tree, noise tolerance) with theoretical guarantees.

Modeling Design Requirements. We first formally model the design requirements of the Internet tree into its definition (termed *IPTree* to avoid confusion): An *IP-Tree* T_P over the IP address hierarchy is a tree whose nodes are prefixes $P \in \mathcal{P}$, and whose leaves are each associated with a label for prediction (e.g., a label may be "0-20ms"). An IPtree is thus a decision tree for IP ad-

dresses $\mathcal{I}$: an IP address i gets the label associated with its longest matching prefix in P. We define the *size* of an IPtree to be the number of leaves needed when it is represented as a binary tree. We define an *adaptive k-IPtree* to be an IPtree that can (a) contain at most k leaves, (b) grow nodes over time, and (c) change the labels of its leaf nodes, and (d) reconfigure itself occasionally. Fig. 1 shows an example IPTree with 4 leaves; each leaf is labeled with the latency range associated with its subtree. Our case studies in Sec. 5 show how all of these operations are useful to maintain an accurate Internet tree.

We need to learn an IPtree with high predictive accuracy, as the accuracy reflects how well it models the data. However, standard decision tree algorithms do not meet many of our design requirements; e.g. most learning algorithms assume that data originates from a fixed distribution; however, we cannot make such an assumption as our tree needs to be able adapt its predictions quickly when there are changes in the input data stream. Most decision tree learning algorithms also do not operate on a data stream – they require multiple passes over the data. The properties required of our IPtree learning algorithm from Section 2 are instead naturally captured in the mistake-bound model of learning [5, 6, 8] and so we build on this model for BirdsEye. For visualization purposes, it is sufficient for the tree to predict the latency within an appropriate range, so we split latency into a number of pre-defined categories and require the tree to predict the right category. [1]

Algorithm Sketch. In the learning model of [8], the algorithm is given an IP address to predict on, makes a prediction, and then is given the correct label to update its internal k-IPtree. At a high-level, the algorithm involves all parent prefixes of an IP i in current IPtree in both steps, i.e., making a prediction for i, as well as in updating itself (i.e., learning). The key aspect of the algorithm is to decompose the main prediction problem into 3 subproblems, which can be treated independently: (a) deciding the prediction of each individual parent prefix, (b) combining the parent prefix predictions by deciding their relative importance, and (c) maintaining the tree structure so that the appropriate subtrees are grown and the unwanted subtrees are discarded. The algorithm casts these subproblems as instances of experts' problems [4, 6], a well-explored area in online learning.

We make 2 major changes to extend this binary classification algorithm to operate on a continuous (but categorized) range of latency values. First, each parent prefix now predicts from m categories instead of 2 using the weighted majority algorithm with shifting targets [6] (instead of shifting experts' algorithm in [8]) – this is another experts' algorithm that allows nodes to shift their predictions between categories over time. Second, we penalize incorrect predictions as a function of how far away they are from their respective true latency values; this way, subtrees with latency categories that are farther apart are grown preferentially, all else being equal.

[1] In this paper, we restrict our problem to predicting latency from a set of pre-defined categories. It is possible also to infer the categories automatically by making multiple passes on the data, but we do not consider this extension in this paper.

4 Visualization

Our visualization makes it easy for operators to detect performance changes and determine where in the Internet they occurred. Figure 2 shows an example of an IPtree displayed in BirdsEye. Each dot is a tree node, which represents a specific IP prefix. There are four relevant properties of each node:

- **Node size** is proportional to the log of the number of RTT measurements that it represents. For example, A has 14,000 measurements whereas B has 112.
- **Node color** corresponds to its predicted RTT; green represents low RTT while red represents high RTT. For example, A has a predicted RTT of 0-20ms whereas B has a predicted RTT of 100-200ms.
- **Distance from center** corresponds to prefix length; shorter prefixes are closer to the center of the circle, while longer prefixes are closer to the edge. For example A is a /6 whereas B is a /24.
- **Angular location** is chosen with the IP interpreted as an integer i, i.e, $\frac{360i}{2^{32}}$. For example, A is a prefix of 128.0.0.0 whereas B is a prefix of 142.0.0.0.

The first two properties make it easy to detect changes by giving obvious ques to the size and severity of anomalies. The second two properties make it easy to determine where changes occur by ensuring that the same IP prefix will always appear in the same place in the visualization and that related prefixes are close to each other.

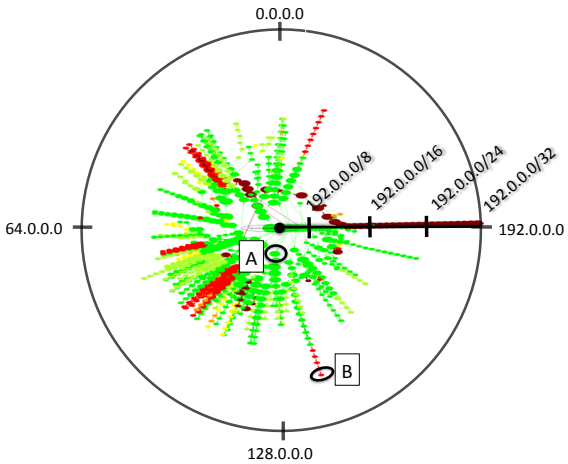


Figure 2: Example of an IPtree displayed in BirdsEye. Each dot is a tree node, which represents a specific IP prefix. Sec. 4 describes how nodes are laid out.

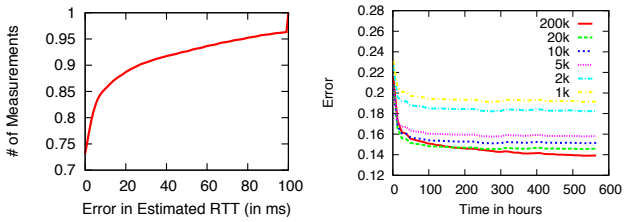


Figure 3: (a) CDF of Error and (b) Misclassified categories as a function of tree size k.

Finally, we must select the parameter k of the IPtree, which determines the number of nodes. Displaying more nodes gives operators more detailed information about network state but may overwhelm them. Displaying fewer nodes presents a more comprehensible picture, but potentially loses important information. Thankfully, the algorithm in Sec. 3 gives us a mechanism to choose the smallest k that doesn't lose much information: We simply choose the smallest k such that the prediction accuracy of the IPtree does not increase significantly with larger k, or drop substantially over time. Sec. 5.1 describes the k selected based on empirical RTT data.

5 Experiments

To demonstrate BirdsEye's utility, we evaluate the accuracy of its IPtree and present simulations and case studies on RTT measurements collected from one node of a large distributed server farm. The node is located near a major metropolitan area in the north-eastern United States. We collected RTT data based on TCP handshake delays using a network monitor on one of the nodes from April 1 to April 20, 2010. 2-3 million measurements are collected each day across all servers at that node.

We implemented BirdsEye with about 3000 lines of C++. Our current unoptimized implementation takes less than 1 minute to generate the IPtree and corresponding visualization for each node in the server farm. Thus, when integrated with an ongoing feed of RTT measurements, BirdsEye can generate near real-time visualizations of network-wide RTT performance.

As discussed in Sec. 3, we split the latencies into categories: $<$20ms, [20ms-40ms), [40ms-60ms), [60ms-80ms), [80ms-100ms), [100ms-200ms), and $\geq$200ms. The categories reflect user perceived performance differences — e.g., an RTT increase from 10ms to 40ms is more noticeable than one from 110ms to 140ms. Nodes change from green to yellow to red as RTT increases.

5.1 Accuracy Evaluation

We now describe our algorithm's accuracy in predicting RTT categories. We note that our goal is not as much to demonstrate a highly-accurate RTT estimation technique, but rather, to show that the tree computed by our algorithm is accurate enough to use for inferring performance via the visualization.

Fig. 3(a) shows the error in estimating the latency category (this is computed as the difference between the actual RTT and the category) for $k = 20,000$, over the entire data set. We note that 83% of RTTs are estimated within 5ms of their category, and 90% are estimated within 20ms, (i.e., to a neighbouring category, at most). Thus, the IPtree's prediction of the latency category is accurate enough for visualizing performance the vast majority of the time.

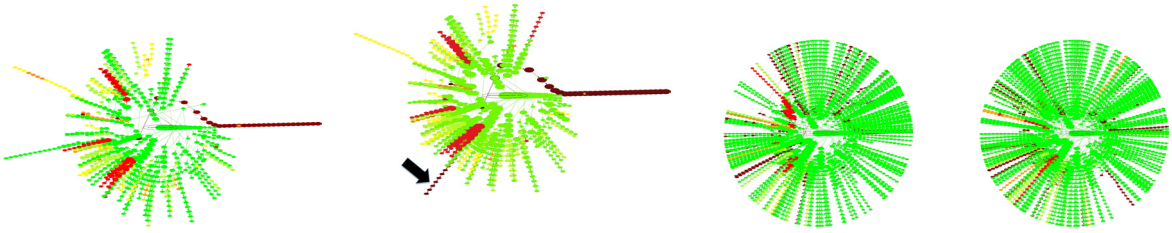
In order to choose an appropriate k for the visualization, we now examine the algorithm's accuracy over time for different k. Fig. 3(b) shows the fraction of misclassified categories over time for k ranging from 1000 to 200,000. Using a tree with $k = 1000$ or $k = 2000$ always produces notably more error than using $k = 5000$ leaves. Increasing k beyond 5000 reduces the error by only by 1% compared to $k = 200,000$. Since visualization is most effective with smaller k, we use $k = 5000$ for the visualization without significant loss in accuracy.

5.2 Injected Anomalies

Next, we use injected anomalies to illustrate Birdseye's ability to visually highlight anomalies. We consider two different anomaly scenarios: one with local performance impact, and another with network-wide impact.

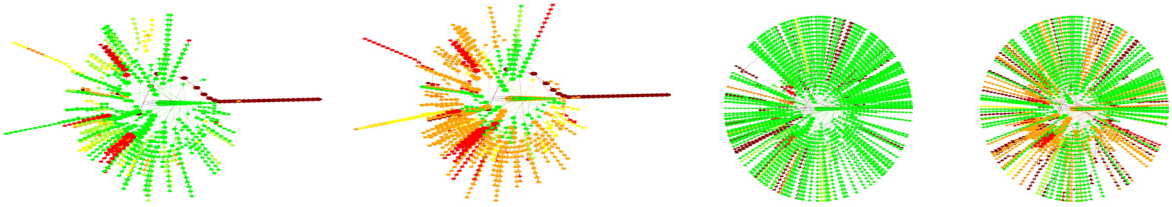
We first consider a class of anomalies that is difficult to detect with traditional methods. Consider a small prefix (e.g., a /24) that is not advertised by itself in BGP (because it is always a part of a larger advertised prefix), and thus would not likely be discovered by examining only advertised BGP prefixes. *How well does BirdsEye handle such a scenario?* We select a /24 prefix of a highly active /13 belonging to a major tier-1 ISP, and add randomly generated IP addresses in this /24 with high RTTs (e.g., 100-200ms) into the stream, such that the injected data is no more than 1% of the parent /13's data. Fig. 4(b) shows BirdsEye IPtree for the hour after this injected anomaly – we see a new red spike (highlighted) corresponding to the anomalous /24, which is not present earlier, i.e., Fig. 4(a). Note that finding such an anomaly even if all /24 prefixes are tracked is not trivial: Figs. 4(c) & (d) show the same anomaly inserted into a strawman tree that consists only of /24 prefixes (but is otherwise grown identically as our tree); the anomaly is lost in Fig. 4(d).

We now illustrate how BirdsEye captures a large-scale performance event. We collect the set of all prefixes advertised by a major tier-1 ISP to the server farm node, and add a delay to each IP address in that set during a time interval. This simulates a scenario where a ISP-wide disruption cause traffic flowing to different destinations through that ISP to be rerouted over much longer paths (e.g., cuts in critical bottleneck links as happened in the Mediterranean Sea in 2008) . Fig. 5(a) & (b) shows



(a) Birdseye: Before anomaly (b) Birdseye: After anomaly (c) Strawman: Before (d) Strawman: After

Figure 4: Injected anomaly: high latency in a /24 prefix block. (a)-(b) show BirdsEye before & after the anomaly, which clearly reveals the /24 block. (c)-(d) show a strawman tree of /24 prefixes, but here the anomaly is lost.



(a) Birdseye: Before (b) Birdseye: After (c) Strawman: Before (d) Strawman: After

Figure 5: Injected ISP-wide performance event: (a)-(b) show BirdsEye before & after the anomaly, which dominates the tree. (c)-(d) show a strawman tree of /24 prefixes, where the anomaly is somewhat less visible.

how this anomaly creates a big visual impact – nearly the entire tree changes in color between the two trees. The strawman tree also changes its color between Fig. 5(c) & (d), but it is less noticeable. Thus, Birdseye is able to visually highlight the impact of a large anomaly as well.

5.3 Real Case Studies

We now present examples of real RTT anomalies discovered using BirdsEye. To illustrate these, we use snapshots of the BirdsEye IPtree at different hours of the day as well as the high-RTT subtrees in Fig. 6.

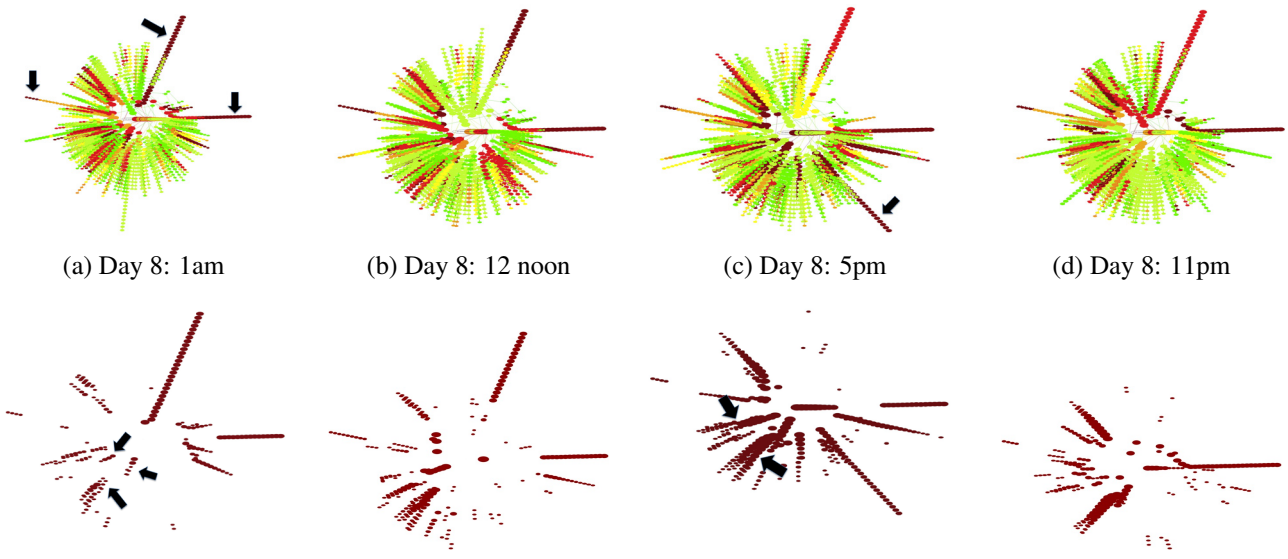
Case Study 1: Consistently Poor Performance. Our first example focuses on parts of the IPtree that always have high RTT (i.e., always appear red). Each snapshot shows 3 consistent long spikes of high RTT in the IPtree (highlighted in Fig 6(a)). On examining these prefixes, we found that they correspond to the management nodes of the distributed server farm located on the West Coast, so their high RTT is not particularly of concern to the server farm's operation. We validated that the IPs do indeed have high RTT by examining the data — $73-81\%$ of the RTTs exceed 90ms, thus justifying their presence.

In addition, there are permanent red areas in all snapshots that are larger prefixes which range from /12 to /18 blocks (highlighted in Fig 6(e)). Recall that larger prefixes are closer to the IPtree's center. Inspection reveals that those prefixes belong to a cellular carrier,[2] and shows that $56-72\%$ of the corresponding RTTs in these prefix blocks are over $80ms$. While high RTTs on wireless carriers are expected, this highlights that those wireless users may have issues accessing latency-sensitive content stored on the server farm.

[2]This carrier is not affiliated with the authors.

Case Study 2: Occasionally Poor Performance. Our second example explores a part of the IPtree that also regularly appears, but experiences high RTTs only occasionally, highlighted in Fig. 6(g). Note that this region shifts from green in the early hours (e.g., Fig. 6(a)) to yellow/red during the busier hours. This is also especially visible from the high-RTT subtrees – these prefixes do not appear in Fig. 6(e), but appear in Fig. 6(g). Closer investigation revealed that most of these prefixes were access ISPs that seem to show signs of congestion during the evening hours, even while there are other ISPs do not (i.e., stay green). Detailed analysis of the measurement data showed that in these access networks, around 40% of the RTTs increased by over $40ms$! Finding the set of all these ISPs using traditional tools, which plot performance per ISP, would have been extremely tedious.

Case Study 3: Anomalous IP block. Our last example shows how BirdsEye may aid in finding an anomaly that may otherwise be lost in noise. We focus on the prefixes highlighted in Fig. 6(c). Note that this spike is not present in the other 3 hours. When we manually examined the prefixes, we discovered that these prefixes belong to a small ISP in the western US, and the IPs appear for 2-3 hours on 3 different days in our data set. They account for less than $0.02-0.05\%$ of the RTTs in those hours, however, 95% of them exceed 80ms, and about 41% exceed 200ms. Even though the IP block comprises a tiny volume of data, BirdsEye differentiates it from a parent prefix with over 100 times more data, 95% of whose RTTs are *under* 80ms. The geographical location of the IP block suggests that these clients were misdirected at the time, as the server farm has nodes that are geographically closer to this ISP. Before BirdsEye, our



(a) Day 8: 1am (b) Day 8: 12 noon (c) Day 8: 5pm (d) Day 8: 11pm

(e) High RTT Subtree: 1am (f) High RTT: 12 noon (g) High RTT: 5pm (h) High RTT: 11pm

Figure 6: Real Case Studies. Figs. (a)-(d) show an IPtree time series through Day 8, and (e)-(h) show the corresponding high-RTT subtrees. Figs. (a) & (e) highlight regions with consistently high RTTs, (g) highlights a region with diurnal pattern and (b)-(d) highlight a one-time misdirected client at the server farm.

operators did not know that this ISP had been directed to this particular node, nor of its extremely high RTT.

6 Related Work

Visualization has been acknowledged as an important way to understand Internet characteristics [2] but an effective visualization requires a compact representation of the data. We focus here on work related to our representation, the Internet tree. Algorithms for building hierarchical heavy-hitter clusters [3, 9] also summarize traffic characteristics into a small number of prefix clusters; however, our problem differs from these as their goal is typically to identify prefixes with substantial traffic, not differentiate performance characteristics. More closely related are approaches that build optimal aggregates [1,7] over the address space to classify traffic with different characteristics; we build on [8] as it is designed for automatically adapting over changing data streams. Our problem also differs from the classic latency estimation problem in networking research: our goal is not to estimate end-to-end latency between arbitrary hosts, but to differentiate latency performance by prefix.

7 Conclusion

We presented BirdsEye, a visualization tool that enables operators to track network-wide performance between a server farm and its customers. It builds *adaptive decision trees* over the IP address space using recent performance measurements, which group IP addresses with similar performance characteristics and separate those with significantly different performance characteristics. By visualizing these decision trees, BirdsEye shows the performance to the entire Internet, but only highlights the parts that have bad performance at any given point in time. As a case study, we used BirdsEye to visualize RTT measurements for a commercial server farm, and discovered several RTT patterns that operators were unaware of but were keenly interested to know, such as diurnal patterns of poor performance in particular access ISPs and an ISP likely misdirected by the server farm. Our approach is likely to be useful in any application where differences in behaviour depend upon the IP address structure.

References

[1] BEVERLY, R., AND SOLLINS, K. An internet protocol address clustering algorithm. In *SysML* (2008).

[2] BURCH, H., AND CHESWICK, B. Internet watch: Mapping the Internet. *Computer 32*, 4 (Apr. 1999), 97–98.

[3] ESTAN, C., SAVAGE, S., AND VARGHESE, G. Automatically inferring patterns of resource consumption in network traffic. In *SIGCOMM* (2003).

[4] FREUND, Y., SCHAPIRE, R. E., SINGER, Y., AND WARMUTH, M. K. Using and combining predictors that specialize. In *STOC* (1997).

[5] LITTLESTONE, N. Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning 2*, 285-318 (1988).

[6] LITTLESTONE, N., AND WARMUTH, M. The weighted majority algorithm. *Information and Computation 108* (1994), 212–251.

[7] SOLDO, F., MARKOPOULO, A., AND ARGYRAKI, K. Optimal filtering of source address prefixes: Models and algorithms. In *INFOCOM* (2009).

[8] VENKATARAMAN, S., BLUM, A., SONG, D., SEN, S., AND SPATSCHECK, O. Tracking dynamic sources of malicious activity at internet-scale. In *NIPS* (2009).

[9] ZHANG, Y., SINGH, S., SEN, S., DUFFIELD, N., AND LUND, C. Online identification of hierarchical heavy hitters: algorithms, evaluation, and applications. In *IMC* (2004).

Polygraph: System for Dynamic Reduction of False Alerts in Large-Scale IT Service Delivery Environments*

Sangkyum Kim
UIUC
kim71@illinois.edu

Winnie Cheng, Shang Guo,
Laura Luan, Daniela Rosu
IBM Research
{*wcheng,sguo,luan,drosu*}*@us.ibm.com*

Abhijit Bose
Google
abose@google.com

Abstract

In order to avoid critical SLA violations, service providers use monitoring technology to automate the identification of relevant events in the performance of managed components and forward them as incident tickets to be resolved by system administrators (SAs) before a critical failure occurs. For optimal cost and performance, monitoring policies must be finely tuned to the behavior of the managed components, such that SAs are not engaged for investigation of false alerts. Existing approaches to tuning monitoring policy rely heavily on high skilled SA work, with high costs and long completion times. Polygraph is a novel architecture for automated tuning of monitoring policies towards reducing false alerts. Polygragh integrates multiple types of service management data into an active-learning approach to automated generation of new monitoring policies. SAs can only be involved in the verification of policies with low projected scores. Experiments with a trace of 60K monitoring events from a large IT service delivery infrastructure compare methods for threshold adjustment in alert policy predicates with respect to potential for false alert reduction. Select methods reduce false alerts by up to 50% compared to baseline methods.

1 Introduction

Proactive prevention and timely response to failures with minimal operational costs is a major target for service providers in large-scale IT infrastructures. In order to achieve this target, service providers use monitoring infrastructures such as IBM Tivoli 7 [5] and HP ServiceCenter 7 [4], to monitor the performance of the managed components and identify critical events. Such events are forwarded to incident management systems, and actions are taken before Service Level Agreement (SLA) violations occur. The monitoring agents deployed on managed components use pre-defined policies to generate events based on Key Performance Indicators (KPIs) and component execution contexts. Other nodes in the monitoring infrastructure perform event aggregation and incident ticket generation based on policies that aggregate in time and space (i.e., across multiple systems). Eventually, SAs analyze the auto-generated incident tickets, called *alerts*, to prevent or solve failures.

The effectiveness of the monitoring policies in capturing critical events with limited false positives and negatives has impact on the service management costs, including the cost of SA time spent with incident management, and the cost of SLA penalties. Previous work [2] and our observation of large-scale delivery infrastructures confirm the difficulty of configuring the monitoring policies for an effective cost balance because of the complexity of the monitored systems and applications.

This paper addresses the problem of effective configuration of the monitoring policies with an automated approach to dynamically modeling system behavior, generating monitoring policies and deploying them. The novelty draws from the integration of diverse service management content (e.g., incident tickets, system vitals) and operational domains (e.g., customers, clusters) and from the use of machine learning to model system behavior and to assess policy effectiveness. As a result, our system, called *Polygraph*, can automatically reduce the number of false-positive alerts, called *false alerts*, with limited or no impact on the identification of *true alerts*. Thus, Polygraph reduces SA work with both alert handling and policy tuning.

Polygraph builds on two novel principles. First principle is to **learn from SAs**, namely, learn from the resolutions of historical incident tickets handled by SAs about alert instances that can be safely ignored. This principle enables effective assessment of policies without elaborate, resource consuming system analysis. The second

*Partially supported by US National Foundation grant IIS-0905215 and the Blue Waters sustained-petascale computing project under NSF grant OCI 07-25070 and the state of Illinois.

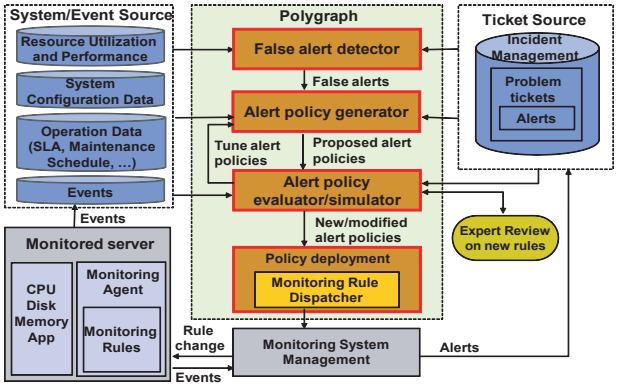


Figure 1: Polygraph system architecture and environment.

principle is to **leverage component similarity in large scale environments** in order to expand the input size for Polygraph learning tasks, and thus improve accuracy. Similarity is also used for policy deployment such that false alerts are reduced even for servers that have not experienced similar events yet, but are likely to experience them in the future. Towards this end, Polygraph uses temporal correlation of system vitals, configuration details, and change operation details.

Figure 1 illustrates the integration of Polygraph within a typical service management infrastructure and its main components. Polygraph has a close interaction with the monitoring infrastructure and leverages data from configuration management databases, repositories of historical system vitals, incident and change management systems, repositories of SLA and maintenance data.

The Polygraph prototype described and evaluated in this paper is focused on the analysis of alerts and generation of new policy. The implementation uses the IBM Tivoli policy specification language. However, our proposal does not depend on a specific monitoring technology, and can be extended to any IT service delivery environment. The evaluation compares several approaches for generation of new monitoring policies., and it is based on over 60K monitoring events, and related incident tickets and system vitals from a large IT service provider.

Overall, this work makes several contributions: (1) Design system architecture for continual refinement and assessment of policies based on integrated exploitation of diverse service management data, (2) Propose techniques for identification of false alerts by mining historical incident resolutions, (3) Propose techniques for generation and assessment of new monitoring policies that can significantly reduce the volume of false alerts.

In this paper, *true alert* identifies an alert instance for which SA intervention is required to solve a critical situation, and *false alert* identifies an alert instance that is cleared without any fix performed by SA, yet involves the SA for system status checks.

Next section discusses related work, Section 3 describes Polygraph architecture and implementation details, Section 4 presents the evaluation of Polygraph prototype. Finally, Section 5 summarizes our results. Refer to [7] for more extensive evaluation results and related work.

2 Related Work

There are several well-known commercial and community-supported platforms for monitoring datacenter and IT infrastructure components, including IBM Tivoli Monitoring [6], and HP OpenView [4]. Most of these products support alert-suppression based on statically specified policies. Our proposal uses policies dynamically generated by mining alert streams and other service management data.

Previous work in the area of incident prevention and resolution in large scale IT infrastructures used incident classification techniques based on performance metrics [1]. Polygraph uses a similar approach of learning from historical service management data in order to discover alert patterns and generate new policies.

In the area of dynamic generation of alert policy, the closest to our work is the approach in [2] for automated and adaptive threshold setting based on application Service Level Objectives (SLOs). This approach is hard to apply in large IT infrastructure due to the complexity of the method and the difficulty to acquire component dependency graphs.

A large body of work relates to false-alert reduction in intrusion detection systems (IDS). The use of data mining methods applied to historical data [9], and multi-stage analysis architectures render our work similar to some IDS solutions [8]. However, the binary IDS alert model (real vs. false attack) is different from the model of performance monitoring alerts, which includes quantified resource usage levels. Thus, this work cannot leverage the IDS methods.

Previous work [3] underlines the difficulty in classifying rare events because traditional learning classifiers are often biased for the most common events. The authors argue that if the events are rare and not too costly, the learning algorithms can do little to improve. When events have high costs (e.g., SLA penalties), a larger number of false alarms must be tolerated. This matches a best practice in IT Service Delivery for management of high-risk SLOs.

3 Polygraph Framework

For a competitive IT service delivery infrastructure, the monitoring infrastructure must minimize the number of false alerts in order to keep operational costs low and

must maximize the number of true alerts in order to prevent SLA failures. This goal can be achieved with detailed analysis and fine-tuning of alert policies. However, even with state-of-the-art tooling, this approach requires substantial SA effort and skills, prohibiting large-scale adoption.

Polygraph, the framework proposed in this paper, specifically addresses these limitations through the automation of monitoring policy evaluation and fine-tuning. The goal of Polygraph is to identify false alerts and design new monitoring policies that lower the occurrences of false alerts with negligible impact of true alerts.

The Polygraph method for false-alert reduction comprises: (1) *learning* the pattern of true and false alerts, (2) *generation* of new policy based on the alert patterns, (3) *assessment* of new policy impact. Figure 1 illustrates the component architecture that implements this method by integration with the overall service delivery infrastructure. Polygraph comprises of four functional components (see Figure 1): **False Alert Detector** performs the analysis of current alert specification effectiveness and false-alert detection. The module distinguishes false and true alerts by learning from SA's assessment of past incident resolutions. Alerts are associated with details about related policy and KPI thresholds, which are used in next stages.

Monitoring Policy Generator performs the generation of monitoring policies based on observed false-alert patterns. Data from similar servers is integrated in order to improve result quality.

Monitoring Policy Evaluator performs the evaluation of newly generated monitoring policies with focus on SLA impact (minimize missed true alerts) and work reduction (maximize eliminated false alerts). Evaluation is based on simulation against historical system vitals and monitoring events. Policies with acceptable balance of SLA impact and reduction move to deployment while others can be passed to SAs for further evaluation.

Monitoring Policy Deployment performs the deployment of new monitoring policies by close interaction with the monitoring infrastructure. Urgency of deployment is assessed base on server profiles, and used for scheduling policy deployment.

The reminder of this section is focused on select elements of Polygraph implementation.

3.1 Policy Model

In general, an alert policy is defined by a user-specified predicate over component KPIs or context parameters. The policy may also include threshold conditions on the event reoccurrence in order to delay alert generation. Polygraph prototype works with three types of policy: [BASIC] IF A; [AND] IF A AND B; and [OR] IF A OR B. Here, A and B are predicate units consisting of one KPI reference and related threshold value. Our methods apply to more complex predicate expressions by conversion to conjunctive normal form.

In the Polygraph prototype, we consider only threshold-based alert policies, which have predicates that refer only to KPIs and threshold values. The analyzed alert traces show that most false-alerts result from threshold-based policies. Sample policies for each type:

- IF ($System.VirtualMemoryUtilization > 90$)
- IF ($NT_Physical\_Disk.Disk\_Time > 80$) AND ($NT\_Physical\_Disk.Disk\_Time <= 90$)
- IF ($SMP\_CPU.CPU\_Status$ = 'off-line') OR ($SMP\_CPU.Avg\_CPU\_Busy\_15 > 95$)

3.2 Learning Alert Patterns

Alerts are identified by analysis of incident ticket details. The alerts-related tickets are automatically generated, and are characterized by a text pattern that may include details about the policy, monitoring event (KPIs and values). Polygraph includes methods for detection of text patterns for alert identification. For each alert, it extracts KPI values, alert policy references, and managed component references. Further, text analysis of incident resolution description helps classify an alert as true alert or false alert.

Given a scope defined by a target alert policy P and a server or group of servers H, Polygraph scans the related historical content to identify alerts and constructs the *true set*, comprising the identified true alerts, and *false set*, comprising the false alerts. A sample approach to capture alert pattern is based on the KPI value patterns, i.e., the ranges of KPI values for each of the true and false sets. Polygraph includes also methods for discovery of time patterns, discussed next.

3.3 Policy Generation with Value Patterns

An alert value pattern is described by the *true value range*, the range of KPI values that corresponds to alerts in the true set. Without loss of generality, we assume a monotonic behavior for KPI values; if a value corresponds to a true alert, than all alerts for higher KPI values are true alerts.

Proposition 1 *For a given alert policy P consisting of one KPI parameter p and its corresponding threshold θ, let T be the true set of P, and $t = min(T)$. If $t > \theta$, P modified to include the threshold t will not generate any false negatives for the given dataset.*

Proposition 1 describes how to tune the threshold value for a BASIC alert policy. If the smallest KPI value t for true alerts is bigger than the original threshold, then

the new threshold value is set to t and no true alerts are missed. False alerts related to KPI values between the old and new threshold are eliminated while those for KPI values above the new threshold remain. As an example, given a policy governing CPU threshold, if all true alerts happen for thresholds greater than or equal to 95%, we can safely raise the original threshold of 90% to 95%. If a false alert occurs for CPU load of 98%, it is not eliminated by the new threshold setting.

3.4 Policy Generation with Time Patterns

To further improve the false alert reduction, Polygraph takes into account the time patterns. Periodic patterns of jobs, like daily, weekly, or even monthly, can significantly affect resource consumption and trigger alerts, false or true. The method comprises the finding of periodic patterns based on learning set and extrapolation of these patterns in the analysis to remaining historical content. In order to limit the risk of missing true alerts by extrapolation, we focus on the periodicity of true alerts, rather than on false alerts.

Given a scope of policy and server(s), periodicity analysis starts with the related true set of the policy and produces a set of periodic time intervals, called *true time ranges*, during which all of the occurring alerts are considered true alerts. Alerts that occur outside the true time ranges are considered false alerts. Periodicity analysis requires specification of a threshold for the width of a true range to mine a set of these true ranges for each alert policy. For example, suppose host H has three true events at 3pm, 4pm, and 8pm. Given a true time range threshold of 1 hour, the analysis results in two true time ranges (2pm-5pm) and (7pm-9pm). Smaller thresholds lead to more false alert detection, but increase the risk of missing true alerts.

3.5 Server-Based Policy Tuning

The typical approach to defining alert policy in service delivery infrastructures is to use the same best-practices policies for all servers with similar installed software and workloads. While this approach yields acceptable alert accuracy during early server lifetime, it will no longer efficient later in server lifetime, across capacity changes (e.g., memory or hard disk upgrades), dynamic re-clustering for workload balancing, and other events. For example, for a server with increase of hard disk capacity from 100GB to 10TB, an alert threshold of 90% utilization will generate alerts when 1TB of disk space is still available.

To address these limitations, Polygraph takes the approach of tuning alert policy thresholds for each server and our experiments show significant benefits. Polygraph does not exclude the tuning of alerts for groups of servers with similar configurations and workloads, which we plan to integrate in the future.

3.6 New Policy Evaluation

In the evaluation phase, Polygraph scans the set of alerts for the current scope (i.e., union of true and false sets) and compares the KPI values against the new policy threshold. The new counters for new true and false alerts are compared with the old counters in order to assess the risk (true alert misses) and benefit (false alert reduction). Installation-specific thresholds on the impact metrics are used to determine if the new policy (1) is highly beneficial with very low impact and should be automatically deployment,(2) has borderline benefits and risks and should be forwarded for SA analysis, or (3) has limited benefits and higher risks, and should be discarded.

4 Evaluation

Our empirical results are based on large and detailed datasets collected from globally distributed production environments serving real clients. We collected 30-day datasets with around 60K events. We divide the datasets of system performance metrics, alerts, and events into six parts (5 days for each) based on their occurred time: older data are to be used for learning purpose, and recent data are for test purpose. To show the effects of training data size on our alert threshold adjustment schemes, we use the first five datasets to make five differently sized training data (datasets of 5, 10, 15, 20, and 25 days) and the last part as test data. By default, the true time range threshold is 1 hour.

The experiments in this section compare various methods for new policy generation with respect the effect of the automatically generated policies to eliminate false alerts. Namely, the experiments present the ratio of false alerts for which the generated policies do not trigger alerts. We refer to the not-triggered false alerts as 'detected false alerts'. The larger the shares of detected false alert, the larger the reduction enabled by Polygraph framework.

4.1 Data Characteristics

In the target service delivery environment, alert policy specification comprises several fields including the target server and predicate descriptions. The threshold values have never been changed since initial deployed (which is typical in most environments).

System KPI values are collected at one-minute intervals and are aggregated over different time windows such as 15 minute, 1 hour, 1 day, and further.

We use two widely deployed alert policies, say P_1 and P_2, in our experiments to show the effectiveness of the Polygraph framework. Table 1 shows their characteristics, including total alert count, share of total alert trace, and distribution across true and false alerts; actual policy

Table 1: Characteristics of select alert policies

Alert Policy	Count	Ratio(%)	True Alerts	False Alerts
P_1	23355	40.48	1026 (4.39%)	22329 (95.61%)
P_2	3344	5.80	1526 (45.63%)	1884 (56.34%)

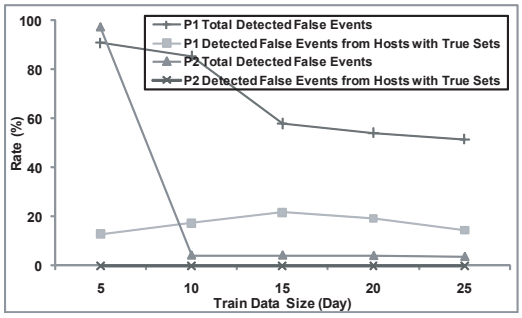


Figure 2: Host-based false alert detection with varying learning set size

expressions are not disclosed for privacy issues. P_1 has a large volume of alerts, of which 95% are false alerts. P_1 is a good example of how Polygraph can automatically and effectively tune an alert policy threshold. P_2 illustrates the case when Polygraph needs expert SA reviews to prevent abuse of automation.

4.2 Basic Threshold Adjustment

The basic threshold adjustment method comprises of policy generation using value patterns and including all servers, i.e., the entire data set is used for assessment of the true set. Both P_1 and P_2 alert policies do not have any gain when applying this method, even if the current policy thresholds are not optimal. This is because servers with same alert policy are not similar with respect to related datasets, as shown by experiments below.

4.3 Host-Based Adjustment

This experiment compares the basic threshold adjustment (i.e., analysis across all servers) with host-based adjustment (i.e., server-based analysis) including only the servers whose training data includes true alerts. Figure 2 illustrates false alert detection rates of P_1 and P_2 as the training set increases. Note that the difference between the plots for each policy indicates the false alert detection rate for servers whose training data do not have any true events. In general, we observe that more false alerts are detected as training set gets smaller. P_1 shows a high rate of false alert detection where as P_2 shows a low detection rate. For P_2, we see that at least 10 days of training data is needed for reliable performance. The spike of P_2 is due to the large number of servers with no true events in the training set.

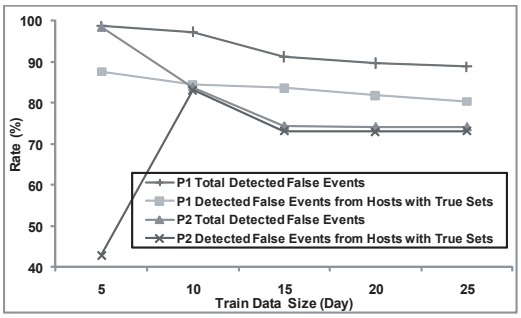


Figure 3: Host and time-based false alert detection with varying learning set size

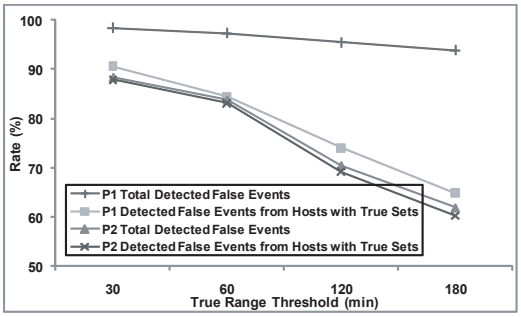


Figure 4: Host and time-based false alert detection with varying true time threshold

4.4 Host and Time-Based Adjustment

This experiment compares the basic adjustment method with a method using the time-based pattern for false alert detection and selection of the servers that experience true alerts. Figure 3 shows false alert detection rates of P_1 and P_2 for the two methods when increasing the training set. In general, the host-and-time-based schema shows higher false alert detection than the time-based schema (see Figure 2). Similarly, the host-and-time-based scheme shows higher false alert detection rates than the host-based scheme. Based on the experiments described above, P_1 can be safely tuned by Polygraph with no human interaction, but P_2 needs to be shown to the system administrator before deployment.

4.5 Impact of True Time Range Threshold

This experiment evaluates the impact of the *true time range threshold* on the rate of false-alert detection when applied to host-and-time-based scheme. The experiment uses the 10-day training dataset and varies the threshold value from 30 to 180 minutes.

The results are illustrated in Figure 4. As the threshold value increases and true time ranges of P_1 and P_2 get larger, the rate of false-alert detection decreases.

4.6 Discussion

Based on our experiments, we identify extensions that can improve the effectiveness of Polygraph in reducing

false-alerts by automated tuning of alert policies.

Leverage operational data for tuning alert policy. The analysis of monitoring event data demonstrates the need to leverage operational data in the tuning of alert policy. For instance, Polygraph integrates operational data that describes when the scheduled maintenance activities are expected to generate significant workload on the managed servers, such as anti-virus scans and backups. Our analysis shows that 20.3% of a customer's alerts are due to virus scan that caused higher CPU usage than the normal state. In order to eliminate these false alerts, one has to exploit operational data and include in the new policies predicates related to the execution context. Another relevant type of operational data includes SLA specifications and attainment statistics. One can exploit SLA information to delay generation of alerts when workload varies significantly but yet the SLAs are unlikely to be missed.

Emphasize more recent history. When long event history is available, such as in typical production environments, false-alert detection is likely to exhibit poor quality if all data samples receive equal weight in the analysis. This is because the detector misses to recognize the most recent trends in the occurrence of false alerts. In order to improve decision quality, a weighted scheme can be employed to emphasize recent input. Weights should be carefully chosen such that the discard of old content does not cause the missing of true alerts.

Scalable policy deployment. Polygraph can generate new policies for a server profile that matches a very large number of servers. In order to avoid the disruption of the monitoring infrastructure, the policy deployment should be staged. The staging order and timeline is based on the assessment of server-specific risks/costs due to delaying the policy deployment.

Impact of change operations. The behavior of a managed system changes over time due to infrastructure changes for the server itself (e.g., hardware, software) or the environment in which the server is running (e.g., workload). As a result, monitoring policy becomes outdated. Integration of service management data that describes change operations, such as new patch/software installation, memory expansion, and subnet change, should be used to trigger analysis for policy tuning and determine the relevant historical content to consider.

Leverage server similarity. Our experiments reveal the potential benefit of grouping similar servers in alert policy tuning. This is helpful in cases when the training dataset collected on an individual server does not have sufficient data points for rare events; grouping similar servers provides a better training dataset, hence better policy tuning. A sample similarity criterion includes the servers in the same web-application cluster, which all have the same server configuration and the same workload characteristics. For more general cases, Polygraph must use a server similarity measure that integrates server resource profiles and workload characteristics.

5 Conclusion

This paper introduces Polygraph, a framework for automated reduction of false alerts in large scale IT infrastructures based on an active-learning approach. Polygraph mines historical incident ticket content to learn from SAs about which alerts are false and to correlate this information with other service management content, such as system vitals and server similarities, to modify threshold-based alert policies and eliminate false alerts. In experiments with real-life traces from a large and diverse service delivery environment, Polygraph performs very well in reducing false alerts while not missing true alerts. Polygraph achieves this by combining host and time-based tuning of alert policies.

In future work, we plan to extend the Polygraph model for automated generation of new policy by including conditions on execution context and methods for automatic identification of durations when maintenance processes generate temporary spikes in resource consumption. Furthermore, for improved quality of false-alert reduction, we plan the development of methods that emphasize recent history.

References

[1] BODIK, P., GOLDSZMIDT, M., FOX, A., WOODARD, D. B., AND ANDERSEN, H. Fingerprinting the datacenter: automated classification of performance crises. In *EuroSys* (2010).

[2] D. BREIGAND, E. H., AND SHEHORY, O. Automated and Adaptive Threshold Setting: Enabling Technology for Autonomy and Self-Management. In *ICAC* (2005).

[3] DRUMMOND, C., AND HOLTE, R. Learning to Live with False Alarms. In *Proceedings of the KDD Data Mining Methods for Anomaly Detection Workshop* (2005).

[4] HEWLETT-PACKARD COMPANY. HP ServiceCenter software. `http://www.openview.hp.com/products/ovsc/index.html` (2011).

[5] IBM CORPORATION. Tivoli Monitoring. `http://www-01.ibm.com/software/tivoli/products/monitor/` (2010).

[6] IBM CORPORATION. Tivoli Software. `http://www-01.ibm.com/software/tivoli/` (2010).

[7] KIM, S., CHENG, W., GUO, S., LUAN, L., ROSU, D., AND BOSE, A. Polygraph: System for dynamic reduction of false alerts in large-scale it service delivery environments. *IBM Research Technical Report* (Apr. 2011).

[8] N. JAN, S. LIN, S. T., AND LIN, N. A decision support system for constructing an alert classification model . *Expert Systems with Applications Vol. 36, No. 8* (Oct. 2009).

[9] SOLEIMANI, M., AND GHORBANI, A. Critical Episode Mining in Intrusion Detection Alerts. In *CNSR* (2008).

Building a High-performance Deduplication System

Fanglu Guo *Petros Efstathopoulos*
Symantec Research Labs
Symantec Corporation, Culver City, CA, USA

Abstract

Modern deduplication has become quite effective at eliminating duplicates in data, thus multiplying the effective capacity of disk-based backup systems, and enabling them as realistic tape replacements. Despite these improvements, single-node raw capacity is still mostly limited to tens or a few hundreds of terabytes, forcing users to resort to complex and costly multi-node systems, which usually only allow them to scale to single-digit petabytes. As the opportunities for deduplication efficiency optimizations become scarce, we are challenged with the task of designing deduplication systems that will effectively address the capacity, throughput, management and energy requirements of the petascale age.

In this paper we present our high-performance deduplication prototype, designed from the ground up to optimize overall single-node performance, by making the best possible use of a node's resources, and achieve three important goals: *scale* to large capacity, provide good *deduplication efficiency*, and near-raw-disk *throughput*. Instead of trying to improve duplicate detection algorithms, we focus on system design aspects and introduce novel mechanisms—that we combine with careful implementations of known system engineering techniques. In particular, we improve single-node scalability by introducing *progressive sampled indexing* and *grouped mark-and-sweep*, and also optimize throughput by utilizing an event-driven, multi-threaded client-server interaction model. Our prototype implementation is able to scale to billions of stored objects, with high throughput, and very little or no degradation of deduplication efficiency.

1 Introduction

For many years, tape-based backup solutions have dominated the backup landscape. Most of their users have been eager to replace them with disk-based solutions that are faster, easier to use (search, restore, etc.) and less fragile. In the past few years, disk-based backup systems have gained significant momentum, and today most enterprises are rapidly adopting such solutions, especially when the data volume is moderate.

One of the most important factors enabling the recent success of disk-based backup is data *deduplication* ("dedupe")—a form of compression that detects and eliminates duplicates in data, therefore storing only a single copy of each data unit. By using dedupe in a disk-based backup system one can multiply the effective capacity by 10-50 times, rendering the system a realistic tape replacement, whose cost is on par with tape-based systems, while also 1) making backup data always available online (for indexing, data mining, etc.), 2) enabling effective remote backups by minimizing network traffic, and 3) reducing client side I/O overhead by eliminating the need to read unchanged, previously backed-up files.

The explosive increase in the amount of data corporations are required to store, however, puts great pressure on the storage and backup systems, creating immediate demand for new ways to address the capacity, performance and cost challenges, and generally increase their overall effectiveness.

The effectiveness of a deduplication system is determined by the extent to which it can achieve three mutually competing goals: *deduplication efficiency*, *scalability*, and *throughput*. Deduplication efficiency refers to how well the system can detect and share duplicate data units—which is its primary compression goal. Scalability refers to the ability to support large amounts of raw storage with consistent performance. Throughput refers to the rate at which data can be transferred in and out of the system, and constitutes the main performance metric.

All three metrics are important. Good dedupe efficiency reduces the storage cost. Good scalability reduces the overall cost by reducing the total number of nodes since each node can handle more data. High throughput is particularly important because it can enable fast backups, minimizing the length of a backup window. Among

the three goals, it is easy to optimize any two of them, but not all. To get good deduplication efficiency, it is necessary to perform data indexing for duplicate detection. The indexing metadata size grows linearly with the capacity of the system. Keeping this metadata in memory, would yield good throughput. But the amount of available RAM would set a hard limit to the scalability of the system. Moving indexing metadata to disk would remove the scalability limit, but significantly hurt performance. Finally, we can optimize for both throughput and scalability, as in regular file servers, but then we lose deduplication. Achieving all three goals is a non-trivial task.

Another less obvious but equally important problem is duplicate reference management: duplicate data sharing introduces the need to determine who is using a particular data unit, and when it can be reclaimed. The computational and space complexity of these reference management mechanisms grows with the amount of supported capacity. Our field experience, from a large number of deduplication product deployments, has shown that the cost of reference management (upon addition and deletion of data) has become one of the biggest real-world bottlenecks, involving operations that take many hours per day, and force a hard limit to scalability.

A lot of the research in the area has focused on optimizing deduplication efficiency and index management, without being able to sufficiently boost single-node capacity: with the current state-of-the-art a single node is limited to a few tens, or hundreds, of terabytes—which is far from sufficient for the petascale. Consequently, scalability has been addressed mostly through the deployment of complex, multi-node systems, that aggregate the limited capacity of each node in order to provide a few petabytes of storage at very high (acquisition, management, energy, etc.) cost. Surprisingly, the problem of reference management performance is largely ignored.

As the rate at which data are generated is rapidly increasing, the pressure for high-performance, scalable and cost-effective deduplication systems becomes more evident. We advocate that single-node performance is of key importance to next-generation deduplication systems: by making the most of a single node's resources, it is possible to build a high-performance deduplication system that will be able to scale to billions of objects. Based on our field experience, we know that such a system would be valuable to a very large number of users (e.g., small/medium businesses) where simplicity is also a top priority. Additionally, we believe that improving single-node performance is essential for multi-node systems as well, since a lot of our techniques can be used to provide more efficient building blocks for these systems, or even collapse them into a single node.

This paper presents a *complete*, single-node deduplication system that covers indexing, reference management, and end-to-end throughput optimization. We contribute new mechanisms to address dedupe challenges and combine them with well-known engineering techniques in order to design and evaluate the system considering all three dedupe goals. *Progressive sampled indexing* removes scalability limitations imposed by indexing, while serving most lookup requests in $O(1)$ time complexity from memory. Our index uses sampling to perform fine-grained indexing, and greatly improves scalability by requiring significantly less memory resources. We address the problem of reference management by introducing *grouped mark-and-sweep*, a mechanism that minimizes disk accesses and achieves near-optimal scalability. Finally, we present a modular, event-driven, client pipeline design that allows the client to make the most of its resources and process backup data at a rate that can fully utilize the dedupe server. As a result, our prototype can achieve high backup (1 GB/sec for unique data and 6 GB/sec for duplicate data) and restore throughput (1 GB/sec for single stream and 430 MB/sec for multiple streams) and good deduplication efficiency (97%), at high capacities (123 billion objects, 500 TB of data per 25 GB of system memory).

The rest of the paper is organized as follows: Section 2 gives a detailed description of the major challenges we had to address. In Section 3 we describe how we address them through our prototype's novel mechanisms, and in Section 4 we present our evaluation results.

2 Challenges

2.1 Indexing

Most deduplication systems operate at the sub-file level: a file or a data stream is divided into a sequence of fixed or variable sized *segments*. For each segment, a cryptographic hash (MD5, SHA-1/2, etc.) is calculated as its *fingerprint* (*FP*), and it is used to uniquely identify that particular segment. A *fingerprint index* is used as a catalog of FPs stored in the system, allowing the detection of duplicates: during backup, if a tuple of the form $<$ FP, location_on_disk $>$ exists in the index for a particular FP, then a reference to the existing copy of the segment is created. Otherwise, the segment is considered new, a copy is stored on the server and the index is updated accordingly. In many systems, the FP index is also crucial for the restore process, as index entries are used to locate the exact storage location of the segments the backup consists of.

The index needs to have three important properties: 1) scale to high capacities, 2) achieve good indexing throughput, and 3) provide high duplicate detection rate—i.e., high deduplication efficiency. Table 1 demonstrates how these goals become very challenging for a

Item	Scale	Remarks
Physical capacity C	C = 1,000 TB	
Segment size S	S = 4 KB	
Number of segments N	$N = 250*10^9$ segs	$N = C/S$
Segment FP size E	E = 22 B	
Segment index size I	I = 5,500 GB	$I = N*E$
Disk speed Z	400 MB/sec	
Block lookup speed goal	100 Kops/sec	Z/S

Table 1: An example system configuration, illustrating some of the challenges involved.

Petascale system. If the system capacity is 1 PB, and the segment size is 4 KB (for fine-granularity duplicate detection), indexing capacity will need to be at least 5,500 GB to support all 250 billion objects in the system. Such an index is impossible to maintain in memory Storing it on disk, however, would greatly reduce query throughput. To achieve a rate of 400 MB/sec, would require the index—and the whole dedupe system for that matter—to provide a query service throughput of at least 100 Kops/sec. Trying to scale to 1 PB by storing the index on disk would make it impossible to achieve this level of performance[1]. Making the segment size larger (e.g., 128 KB) would make deduplication far more coarse and severely reduce its efficiency, while still requiring no less than 172 GB of RAM for indexing.

It becomes obvious that efficient, scalable indexing is a hard problem. On top of all other indexing challenges, one must point out that segment FPs are cryptographic hashes, randomly distributed in the index. Adjacent index entries share no locality and any kind of simple read-ahead scheme could not amortize the cost of storing index entries on disk.

2.2 Reference Management

Contrary to a traditional backup system, a dedupe system shares data among files by default. Reference management is necessary to keep track of segment usage and reclaim freed space. In addition to scalability and speed, reliability is another challenge for reference management. If a segment gets freed while it is still referenced by files, data loss occurs and files cannot be restored. On the other hand, if a segment is referenced when it is actually no longer in use, it causes storage leakage.

Previous work [12, 19] mainly focused on indexing and largely ignored reference management. Some recent work [4, 18] started to acknowledge the difficulty of the problem. But, for simplicity, only simple reference counting was investigated without considering reliability and recoverability. Reference counting, however, suffers from low reliability, since it is vulnerable to lost or repeated updates: when errors occur some segments may be updated and some may not. Complicated transaction rollback logic is required to make reference counts consistent. Moreover, if a segment becomes corrupted, it is important to know which files are using it so as to recover the lost segment by backing up the file again. Unfortunately, reference counting cannot provide such information. Finally, there is almost no way to verify if the reference count is correct or not in a large dynamic system. Our field feedback indicates that power outages and data corruption are really not that rare. In real deployments, where data integrity and recoverability directly affect product reputation, simple reference counting is unsatisfactory.

Maintaining a reference list is a better solution: it is immune to repeated updates and it can identify the files that use a particular segment. However, some kind of logging is still necessary to ensure correctness in the case of lost operations. More importantly, variable length reference lists need to be stored on disk for each segment. Every time a reference list is updated, the whole list (and possibly its adjacent reference lists—due to the lists' variable length) must be rewritten. This greatly hurts the speed of reference management.

Another potential solution is mark-and-sweep. During the mark phase, all files are traversed so as to mark the used segments. In the sweep phase all segments are swept and unmarked segments are reclaimed. This approach is very resilient to errors: at any time the process can simply be restarted with no negative side effects. Scalability, however, is an issue. Going back to the example of Table 1, we would need to deal with N = 250 billion segments. If a segment FP is E = 22 bytes, that would be $I = N * E$ = 5,500 GB of data. If we account for an average deduplication factor of 10 (i.e., each segment is referenced by 10 different files), the total size of files that need to be read during the mark phase will be 55,000 GB. This alone will take almost 4 hours on a 400 MB/sec disk array. Furthermore, marking the in-use bits for 250 billion entries is no easy task. There is no way to put the bit map in memory. Once on disk, the bit map needs to be accessed randomly multiple times. This also takes significant amount of time. One might want to mitigate the poor performance of mark-and-sweep by doing it less frequently. But in practice this is not a viable option: customers always want to keep the utilization of the system close to its capacity so that a longer history can be stored. With daily backups taking place, systems rarely have the luxury to postpone deletion operations for a long time. In our field deployment, deletion is done twice a day. More than 4 hours in each run is too much. In a large production-oriented dedupe system reference management needs to be very reliable and have good recoverability. It should tolerate errors and always ensure correctness. Although mark-and-sweep provides

[1] Our measurements show that even high-end SSDs cannot achieve more than 60 Kops/sec

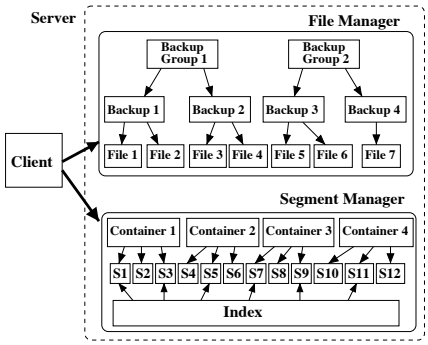


Figure 1: Client and deduplication server components. The server components may be hosted on the same or different nodes.

these properties, its performance is proportional to the capacity of the system, thus limiting its scalability.

2.3 Client-Server Interaction

Even if we solve the indexing and reference management problems, high end-to-end throughput is not guaranteed. An optimized client-server interface is necessary to reap the benefits of deduplication. The typical dedupe client performs the following steps during backup: 1) read data from files, 2) form segments and calculate FPs, 3) send FPs to the server and wait for index lookup results, and 4) for each index miss, transmit the relevant data to the server—otherwise create references to the existing segments. This process may suffer from three different types of bottlenecks. First, reading files from disk is an I/O-bound operation. Second, calculating cryptographic hashes is a very CPU-intensive task, and the client may not be able to compute FPs at the necessary rate. Finally, high latency and low communication throughput may become the main bottleneck for overall performance.

3 Prototype Design

3.1 Goals and System Architecture

We set our performance goals as follows:

- **Scalability**: store and index hundreds of billions of segments.
- **Deduplication efficiency**: best-effort deduplication: if resources are scarce, sacrifice some deduplication for speed and scale.
- **Throughput**: near-raw-disk throughput for data backup, restore, and delete.

To that end, we have implemented a prototype of our scalable duplication system aiming to validate the effectiveness of the proposed mechanisms. Our implementation uses C++ and pthreads, on 64-bit Linux, and it is based on the architecture shown in Figure 1.

The server side component consists of two main modules—the *File Manager* and the *Segment Manager*—that implement all the deduplication and backup management logic.

The File Manager (*FM*) is responsible for keeping track of files stored on the deduplication server. The FM manages file information using a three level hierarchy, visible in Figure 1. The bottom level consists of *files*, each represented by a set of metadata and identified by a file FP, calculated over all segment FPs that the file consists of. The middle level consists of *backups*, that group files belonging to the same backup session. At the top level, multiple backups are aggregated to a *backup group*, allowing the FM to perform coarse-granularity tracking of file/backup changes in the system, so as to assist our reference management mechanism.

The Segment Manager (*SM*) is responsible for the indexing and storage of raw data segments, and may run on the same or a different server than the FM. Segments are stored on disk in large (e.g., 16 MB) storage units, called *containers*. Containers consist of raw data and a catalog which lists all FPs stored in the container. All disk accesses are performed in the granularity of containers. Storing adjacent segments in the same container greatly improves dedupe performance, by reducing container I/O and by improving indexing efficiency (as discussed in Section 3.2.1). The SM also incorporates the dedupe index, and updates it when segments are added/removed.

The client component reads file contents or receives data streams (e.g., data from *tar*), performs segmentation, and calculates segment FPs. After querying the SM index, the client creates references to the existing copies of FPs located in the SM, and initiates data transfers for new FPs. Once a file has been fully processed, the File Manager is updated with file metadata.

Without loss of generality, we use fix-sized, 4 KB segments, for fine-granularity dedupe—although none of the mechanisms relies on this assumption.

3.2 Progressive Sampled Indexing

Most dedupe systems, when performing backup restore, rely on the index—or a similar catalog-like structure—in order to determine the disk location of each segment. This forces the strict requirement for at least one *complete index* containing location information for all FPs, that the system will have to maintain and protect against crashes, corruption etc., because errors cannot be tolerated. If a segment's disk location cannot be determined due to index failure, the whole file or backup gets corrupted. Maintaining such a data structure is a difficult and resource consuming task, that almost certainly impacts system scalability and performance, since the index typically needs to be stored both in memory, for performance, and on disk, for durability.

In order to address the indexing challenges and scale to billions of objects with high performance we had to remove this restriction by introducing *directly locatable objects*: when a file is stored in the system, file segment location information is stored with the file metadata, therefore removing the need to consult the index for the exact location of file segments. For example, if file F consists of segments with FPs A, B and C, stored at disk locations 1, 2 and 3 respectively, F would be represented by the list "$A,1,B,2,C,3$"—instead of just "A,B,C". The increased file metadata size is not a problem, since metadata are stored on disk, while the indexing freedom we get in exchange is extremely valuable.

By decoupling indexing and restore we no longer need to maintain a full index. Instead, we introduce *sampled indexing*, that is based on the observation that given certain amounts of memory and raw capacity, we can calculate the index size, and determine the number of entries that need to be dropped. In particular, if M is the amount of memory available for indexing (in GB), S is the dedupe segment size (in KB), E is the memory entry size (in bytes), and C is the total supported storage (in TB), then we can support M/E billion entries, while the system consists of a total of C/S billion segments. Therefore, if we assume a sampling period T, signifying that we maintain "1 out T" fingerprints in memory, we can define a sampling rate R as follows:

$$R = 1/T = (M/E)/(C/S) = (M*S)/(E*C) \qquad (1)$$

In the example of Table 1, using 22 bytes per index entry, with 4 KB segments and 64 GB of memory for indexing, we can support 11.6 TB of data with a sampling rate of 1 (i.e., a full index). Scaling to 1,000 TB, would require a sampling rate of 0.0116—i.e., insert in the index one out of 86 FPs. Using an 8 KB segment, we could double the raw capacity, or double the rate to $1/43$, sacrificing some dedupe accuracy for higher index density. Increasing the indexing capacity of the system by adding more RAM is rewarded with higher sampling rates (i.e., better dedupe efficiency), while increasing only the storage capacity results in a lower sampling rate, but this is often acceptable, in return for "infinite" system scalability.

3.2.1 *Dedupe efficiency: pre-fetching and caching.*

Since "1 out of T" FPs is inserted in the index, index hits—and, consequently, dedupe efficiency—would be reduced by a factor of T. However, when a lookup operation hits on a sampled FP (also referred to as a "hook"), we locate the container it belongs to and pre-fetch all FPs from that container's catalog into a memory cache. It has been shown [19] that the likelihood of subsequent lookups hitting on the FP cache is high, due to spacial locality: if hook FP A was followed by dropped FP B, then it is very likely that A and B will reappear in order in the future, in which case A will have seeded pre-fetching of its container catalog, resulting in a cache hit for B.

Container catalog pre-fetching can be extremely effective in improving the deduplication efficiency of a sampled index. However, pre-fetching introduces a minimum sampling rate: at least one FP per container (e.g., the first FP stored in the container) must be in the memory index as a hook, in order to seed pre-fetching. Because of this, if container size is K MB, then $R \geq R_{min} = S/(K*2^{10})$ and, subsequently, scalability is no longer "unlimited": the maximum supported capacity is now $C \leq (M*K*2^{10})/E$. For 4 KB segments and 16 MB containers, at least 1 out 4096 FPs needs to be sampled, and with 64 GB of RAM, as in the example of Table 1, $C \leq 47,662$ TB—which is still very high.

Deduplication efficiency. Although the combination of sampling and FP pre-fetching can often yield up to 100% duplicate detection, random eviction of cache entries may reduce deduplication. Using a simplified model we can estimate the dedupe efficiency of the system. Each container catalog contains at most $(K*2^{10})/S = 1/R_{min} = T_{min}$ entries. If we want to achieve deduplication efficiency $f\%$, and we suffer x misses from one container, then:

$$f/100 = 1-(x/T_{min}) \Rightarrow x = T_{min}*(1-(f/100)).$$

If a particular container suffers one eviction during a large time frame (most likely scenario, especially when LRU is used), then all x misses will fall between two consecutive hooks hitting on the index, and therefore:

$$T = 1/R = x+1 \Rightarrow T = T_{min}*(1-(f/100))+1 \Rightarrow$$

$$\Rightarrow (E*C)/(S*M) = T_{min}*(1-(f/100))+1 \qquad (2)$$

Using Equation 2 we can calculate that in the example of Table 1, with 64 GB of memory, the deduplication efficiency will be $f = 97.9\%$. Alternatively, for a given target dedupe efficiency, we can calculate the necessary values to achieve it: for example, if we want $f \geq 95\%$, and given E, C and S, the amount of memory required is $M \geq 26.7$ GB.

3.2.2 *Progressive Sampling.*

A simple, yet important, optimization to sampled indexing is based on the observation that Equation 1 is using the total storage capacity of the system, and, therefore, calculates the value of R_{tot}, required to support all C/S billions of objects. However, at any given time, only the amount of data that are actually stored in the system need to be indexed, which allows us to utilize a *progressive*

sampling rate that calculates R using the amount of storage *used*, as opposed to the maximum raw storage. Initially we set $R = 1$, and gradually decrease it as more storage gets used. In our working example, with 64 GB of RAM, $R = 1$ can index 11 TB of storage. As we approach the 11 TB limit, we can set $R = 0.5$ and down-sample the index (e.g., drop index FPs with $FP \ mod \ 2 \neq 0$), thus doubling the indexing capacity. Eventually, as usage approaches 1,000 TB, R will converge to $R_{tot} = 0.0116$.

3.2.3 Implementation

The index and cache have been implemented in C++ using a highly parametrizable hash table design, which we call *dhash*, optimized for high performance and efficient memory usage. The M GB of memory available for indexing are divided to fixed size buckets (1 KB by default), allowing us to have a maximum of $Y = M/bucket\_size\_in\_KB$ millions of buckets. No pointers are used in a dhash structure, and all operations use offsets, allowing us to 1) perform custom memory management (bucket slab allocator), 2) get memory savings by replacing each 8-byte pointer with 6 bytes of offset data, and 3) make the dhash easily serializable (e.g., when checkpointing to disk at system shutdown).

If a dhash is used at the role of the index, we aim to accommodate as many sampled FP entries as possible. We utilize 2^b buckets for the hash table, where $b = log_2(Y * 2^{20}) - k$. The system parameter k determines the number of buckets reserved for collision handling. Each index entry contains a partial FP (since the b least significant bits of the FP are encoded in the hash table position), and the container number the FP belongs to. For simplicity we use 128-bit MD5 (which is not strong enough for production, but adequate for our testing purposes), leading to a typical entry size of 18 bytes[2]. Each index dhash also utilizes a Bloom filter, to avoid unnecessary lookup operations, which greatly improves performance.

A cache dhash is optimized mainly for performance: it will use all buckets for the hash table, and handle collisions by running a cache eviction algorithm. A cache dhash can employ one of three eviction policies when collisions for a particular bucket Q occur: *Immediate eviction* will empty Q, and consider all the containers of Q's previous entries as evicted from the cache. This policy is very fast since it performs lazy eviction of FPs, allowing for subsequent lookups to hit on those entries. On the downside, this policy penalizes multiple containers at once. *Eviction by threshold* is similar to immediate eviction, but the containers whose entries are being removed from Q will not be considered as evicted until a certain percentage of their total entries has been removed from all cache buckets. This imposes less of a penalty to containers with entries in Q, but may lead to poor deduplication if the threshold is high, since a particular container may not be pre-fetched even though many of its entries have been evicted. *Container LRU* will evict the entries of the least recently pre-fetched container. If that does not free up space in Q, the process is repeated. Although this is the policy that yields maximum dedupe efficiency, it is also the one with the most overhead. Our default policy is immediate eviction, which provides good deduplication efficiency, and performance only slightly lower than eviction by threshold.

In order to provide high dedupe efficiency after system reboots or crashes, we must ensure that a relatively recent index checkpoint is stored persistently[3]. Bucket change-tracking combined with our pointer-free implementation make checkpointing efficient (only a few seconds per checkpoint). Our current policy creates checkpoints every few minutes, and on system shutdown.

SSD indexing. Although sampling provides an efficient way around scalability restrictions imposed by memory limitations, we wanted to also provide a way to improve scalability even with modest amounts of memory, and without having to resort to very low sampling rates. To that end we have also implemented a (persistent) SSD-based version of our sampled index. Sampled fingerprints are stored on sorted SSD blocks and all available memory is used for three performance optimizations: 1) create an SSD summary data structure *SSD_sum*, 2) maintain a Bloom Filter for the SSD index, and 3) maintain an FP cache of pre-fetched containers—similar to that used for the memory index. The *SSD_sum* data structure keeps track of the first FP in each of the SSD's (sorted) blocks, thus allowing us to perform any lookup with at most one SSD block read: when a *lookup(X)* operation is performed, X may be found in the cache, or it may be found by reading the SSD block i, where $SSD\_sum(i) \leq X < SSD\_sum(i+1)$. The SSD index is read-only, eliminating the need for shared locking during accesses. All SSD index updates are cached and logged. Eventually, index updates are performed in batches (and with the SSD exclusively locked): for our 128 GB SSD a full update takes less than 9 minutes, and we can afford to update the SSD many times per day.

3.3 Grouped Mark-and-Sweep

The challenge in reference management, as discussed in Section 2.2, is to ensure reliability while ensuring that the reference management mechanism is also both scalable and fast enough to keep up with the backup speed. A mark-and-sweep approach is very reliable, but offers

[2] With a stronger 160-bit hash, the entry size becomes 22 bytes.

[3] Notice that even if we lose all index index entries, correctness is preserved.

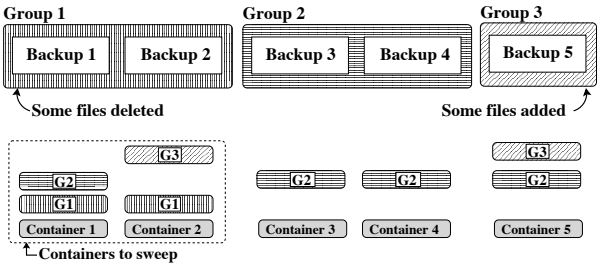


Figure 2: Example illustrating the scalability of grouped mark-and-sweep.

poor scalability because it needs to touch every file in the system. To address this challenge we propose the *grouped mark-and-sweep* (GMS) mechanism, which is reliable, scalable, and fast. The key idea is to avoid touching every file in the mark phase and every container in the sweep phase. GMS achieves scalability because its workload becomes proportional to the changes—instead of the capacity of the system.

The operation of GMS is based on change-tracking within the File Manager. As presented in Figure 1, the File Manager keeps track of files, backups, and backup groups. A file can be a regular file, a database backup stream, an email, etc. A backup is a set of files, e.g., all files under a set of directories. The creation and contents of backups are in the control of the user.

Backup groups aim to control the number of entries that GMS needs to manage, and are created and managed by the File Manager. When backups are small, we aggregate multiple small backups to one bigger backup group. The File Manager tracks changes to each backup group, and for each changed backup group, it further tracks whether files have been added to or deleted from it. During a GMS run, the following steps take place:

1. **Mark changed groups.** Only mark the changed backup groups and do nothing for unchanged backup groups. As an example in Figure 2, assume that File Manager's change tracking shows that, since the last GMS cycle, we deleted some files from group Group1, added some files to group Group3, and made no modifications to Group2. In this case we only need to touch files in backup groups Group1 and Group3. Usually, most backup groups (e.g., Group2) are not changed and files in those groups don't need to be marked. The mark results of G1 and G3 are recalculated by traversing all files in Group1 and Group3 and recalculating G1 and G3 for all containers that have segments used by those files. A group's mark results, say G1, is a bitmap implemented as a file for each container.
2. **Add affected containers to the sweep list.** Only containers used by groups that have deleted files need to be swept because only those containers may have segments freed. In the example of Figure 2, Group1 has files deleted and it has used containers 1 and 2. So we put these two containers in the sweep list. The segments in other containers are either still referenced by files in the unchanged groups (say Group2), or referenced by new files in new groups (say Group3).
3. **Merge, sweep, and reclaim freed space.** For each container in the sweep list, we merge the mark results of all groups using that container. If a segment is not used, it can be reclaimed. In the example of Figure 2, for Container 1, we merge (the old) G2 and (the new) G1, to determine potentially unused segments. Similarly, we merge (the new) G3 and (the new) G1, to determine potentially unused segments in Container 2.

As it becomes clear from the example of Figure 2, GMS provides two important scalability benefits. First, old mark results (e.g., G2) can be reused, without having to re-generate them in every mark-and-sweep cycle. Each set of mark results is stored and reused in the future, making the mark phase scalable by avoiding to touch the majority of the unchanged backup groups. Secondly, unlike conventional mark-and-sweep where all the entries are swept to determine the unused entries, in GMS we know which containers have reference removal operations, and the system only needs to sweep that subset of containers. Therefore the majority of containers in the system are usually not touched in the sweep step.

One drawback of GMS is that a group needs to be remarked even if just one file has been deleted from it. Fortunately the overhead is surprisingly small: segments can be marked at a rate of 26 GB/sec. Since most bitmaps are not changed, there are little work in the sweep phase.

Overall, GMS makes mark-and-sweep scalable by only touching the changed objects, while maintaining the reliability of mark-and-sweep. If errors occur, the whole process can start over and all operations are idempotent. Finally, the mark results (e.g., G1 and G2 for Container 1) serve as a coarse reference list for segments in the containers. When data corruption occurs in a container, the mark results can give us a complete list of backup groups that use that particular container. This limits the set of affected files significantly, and greatly enhances recoverability. Otherwise, we would need to go through all files in the system to determine which files are using that container.

Discussion. An interesting issue related to reference management is concurrent reference updates (data deletion) and data backup. In the example of Figure 2, Backup 5 may still be active when it gets marked, and after all changed backup groups are marked, GMS determines that segment x can be deleted. If Backup 5 uses

x between the time Backup 5 was marked and the time that GMS deleted segment x, data loss will occur as a backup uses deleted/non-existent segments. HYDRAstor [4] uses a read-only phase to freeze the system while updating segment reference counts. In practice, the viability is dubious. On a busy system, there are always some active backups. It is very unlikely to find a time window when the system can be frozen.

Our system uses an in-memory protection map to address this problem: after GMS begins, all segments used by current active backups are protected by storing their segment fingerprints in a protection map in memory. GMS only deletes segments whose fingerprint is not in the protection map. This way GMS can be certain that segments in use will never get deleted. The protection map grows while GMS is running and gets deleted once GMS completes. This is another reason why GMS needs to be fast enough to prevent the protection map from using too much memory. To mitigate the time spent in GMS, and limit the growth of the protection map, GMS can be done more frequently.

3.4 Client-Server Interaction

Even with high-performance server components, it is impossible to achieve high throughput, unless the client is able to push data to the server at a high-enough rate. To that end, our client component is based on an event-driven, pipelined design, that utilizes a simple, fully asynchronous RPC implementation.

Our RPC protocol is implemented via message passing over TCP streams or system IPC mechanisms (e.g., named pipes), depending on whether communication is remote or local. The TCP implementation utilizes multiple TCP connections to keep up with the throughput requirements. All RPC requests are asynchronous and batched in order to minimize the round-trip overheads and improve throughput. A client can register different callback functions for each type of RPC. The callback functions are used to deliver the RPC results to the caller as they become available.

Based on our asynchronous RPC protocol, we have implemented an event-driven client pipeline, presented in Figure 3, where each backup step is implemented as a separate pipeline stage.

First, the reader thread R receives the backup schedule, reads large chunks of data (e.g., 256 segments), and enqueues requests to the hash queue HQ. The hashing thread H dequeues requests from HQ, performs segmentation for each data chunk, and calculates FPs. Calculating cryptographic hashes is a computationally expensive operation, and, in order to fully utilize multiple CPU cores, H employs n MD5 worker threads $(H_1, H_2, \ldots, H_n)$ that calculate FPs asynchronously. Once a chunk's segment FPs have been calculated, callback function $CB1$

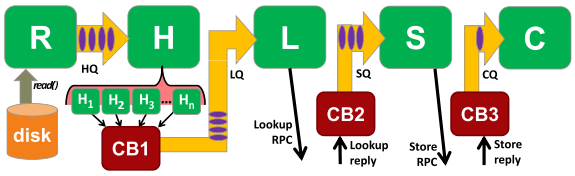


Figure 3: Client pipeline, consisting of five main event-handling threads connected using queues.

enqueues the updated request to the lookup queue LQ.

The lookup thread L receives requests from LQ and issues one single, batched, asynchronous lookup RPC to the server, incurring a single RPC round-trip for all 256 FPs. Callback function $CB2$ delivers the RPC reply and creates references to the containers of the FPs that were found on the server. If one or more FPs were not found, CB2 enqueues the updated request in the store queue SQ.

The store thread S receives requests from SQ, and sends raw data blocks to the back-end through one single, batched, asynchronous RPC. Callback function $CB3$ ensures that the write operation was successful, and forwards the last request for each file to the close queue CQ.

Finally, close thread C, receives the final request from CQ, performs cleanup, calculates file metadata, and updates the File Manager.

Client queues allow us to better understand system behavior. For instance, on a client with low hash calculation throughput, we can observe HQ to be full most of the time, while low network performance will lead to LQ and SQ being mostly full. In such cases, more than one threads can be used for each pipeline stage. By using two store threads, for example, we can consume requests from SQ at a higher rate.

4 Evaluation

Our main test-bed is an 8-core Xeon E5450 at 3 GHz with 32 GB RAM, running Linux. Our 24 TB disk array consists of 12 disks, 2 TB each, and uses RAID 0[4] to stripe all physical disks to a single logical volume.

We used two main data sets for testing. Our synthetic data set consists of multiple 3 GB files, each with globally unique data segments. Our second data set consists of virtual machine images, which are a very common real-world enterprise use-case, that takes advantage of deduplication. We use a VMware "gold" disk image ($VM0$), hosting a Microsoft Windows XP installation, and created three additional versions of it ($VM1$, $VM2$, and $VM3$), each with incremental changes: VM1 is VM0 with all Microsoft updates and service packs, VM2 is

[4]RAID 0 is not recommended for a high-availability system, but we used it to achieve maximum performance and mitigate the disk bottleneck—thus emulate a high-end array.

VM1 with a large anti-virus suite installed, and VM3 is VM2 after the installation of various utilities (document readers, compression tools, etc.). This data set aims to measure the "real-world" dedupe performance of our system, using a file type of great importance for the enterprise.

For both data sets we configured the system to use a sampling rate of $R = 1/101$, which is low enough to stress the system. For the synthetic tests performed on our current test-bed, the index uses 25 GB memory to hold 1.23 billion FPs. With a sampling rate of 1/101, this is equivalent to a full index of 124 billion FPs, or 500 TB of raw storage—given that our segment size is 4 KB[5].

4.1 Throughput

4.1.1 Backup Throughput

Index throughput. Before performing any macro-benchmarks, we used micro-benchmarks to ensure that the index can support our goals—e.g., in the example of Table 1, at least 400 MB/sec. In all the micro-benchmarks the index could easily handle the desired rates: insert/lookup/remove cost does not exceed 7,619/12,020/16,836 cycles, respectively, even when index occupancy is more than 97%. For instance, on a 3 GHz CPU, and in the worst-case scenario where all incoming FPs exist in the system (and the Bloom filter is of no help), the index can sustain a backup rate of around $975 * T$ MB/sec, where T is the sampling period. For our test configuration, $T = 101$, and the index can sustain a rate of about 98.5 GB/sec.

Unique data: baseline vs. prototype. Figure 4 shows the backup throughput using the synthetic data set. We vary the number of concurrent backups, in steps of 1, 4, 16, 32, 64, 128 and 256, in order to evaluate the system's capability for concurrency. For consistency, all backups consist of multiple 3 GB files that add up to 768 GB.

The unique data throughput test aims to measure the prototype's behavior in the absence of duplicates. Unique data can be significant when a client performs the initial backup or a lot of changes have been made. This test stresses the disk and the network systems as large amounts of data need to be transferred.

To get a sense of the performance of raw hardware, we first measured a baseline throughput. The baseline throughput of the disk array ("Baseline" in Figure 4), is measured by writing the same synthetic workload to the file system. For a single backup, the baseline throughput is around 1 GB/sec. This is the maximum throughput of the storage system. The baseline throughput quickly drops to around 300 MB/sec for storing multiple backups concurrently because disk contention increases with the number of concurrent backups.

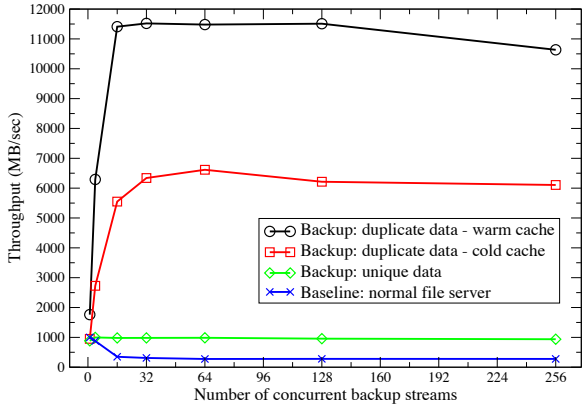


Figure 4: Aggregate throughput for our synthetic data set, with varying number of concurrent backups. Our system is capable of 6 GB/sec for duplicate data backup, and close to 1 GB/sec for concurrent backups of unique data. Dedupe efficiency is 97%, and we support 200 TB storage for every 10 GB of system memory (500 TB for 25 GB in this test).

Backing up the same data set ("Unique data" in Figure 4) using our prototype achieves a steady throughput of about 950 MB/sec as we scale to multiple concurrent backups, which is significantly better than the regular file server. This is mainly because our prototype performs segmentation on all incoming data, and manages the serialization of containers to disk (regardless of content source), therefore decreasing concurrent disk accesses.

Duplicate data backup throughput. After backing up the unique data workload using our prototype, we backup the same files again ("Duplicate data - cold cache" line in Figure 4). This time, all segments are duplicates, and we aim to observe how our prototype performs when it only needs to reference existing data, instead of physically storing new data. This test mainly stresses the index lookup and disk pre-fetching operations.

Initially, for low levels of concurrency, the penalty for small random disk reads, for container FP catalog pre-fetching, dominates performance. Throughput improves steadily as we increase the level of concurrency and duplicate elimination pays off, with aggregate disk throughput reaching over 6.6 GB/sec for 64 concurrent backup streams. When disk accesses are already random, concurrent access doesn't introduce more randomness. On the other hand, concurrent accesses can fully utilize every disks in the disk array. Thus the aggregate throughput increases. After 64 concurrent streams, the disk ar-

[5]Testing our system with a configuration that supports a raw capacity of 500 TB per node may seem inadequate at first. One should keep in mind, however, that 1) We are stressing the system by using 4 KB segments. Most systems use significantly larger segments, leading to higher raw capacities. 2) This is *single-node* capacity with only 25 GB memory for indexing. As such, it is higher than that of most systems we know of (as presented in Section 4.4). Unfortunately we don't have access to servers with more memory or larger disk arrays so as to test higher capacities.

Backup streams	Unique data	Duplicates (cold cache)	Duplicates (warm cache)
1	840 (-4.9%)	699 (-26.4%)	1,989 (12.9%)
4	992 (-0.5%)	2,556 (-6.3%)	6,326 (0.6%)
16	999 (1.9%)	4,802 (-0.2%)	11,992 (5.1%)
32	985 (0.3%)	6,420 (1.3%)	12,134 (5.3%)
64	984 (-0.2%)	6,621 (0.1%)	11,865 (3.3%)
128	988 (3.2%)	6,315 (1.6%)	11,755 (2.1%)
256	955 (1.9%)	6,041 (-1.1%)	11,946 (12.3%)

Table 2: We repeated the experiments of Figure 4 using the SSD index. Results are in MB/sec. The percentages in parentheses show how much faster/slower the SSD index is from the memory index.

ray's capacity for pre-fetching is saturated and mild effects from concurrency overhead (index/cache locking, disk accesses etc.) are becoming obvious: duplicate data backup throughput falls to 6 GB/sec and remains mostly constant.

To verify our conjecture that duplicate data backup throughput limitations are mainly due to disk bottleneck (container FP catalog pre-fetching) instead of CPU, we backup the same files a third time immediately after the second backup. In this case, many FPs are already in the cache and fewer disk pre-fetches will be necessary. The throughput is shown as "Duplicate data - warm cache" in Figure 4. First we observe that overall throughput is much higher, reaching 11.5 GB/sec at around 16 streams, confirming that the bottleneck in our previous tests was in the disk random access performance, which determines the duplicate backup throughput. Additionally, we observe that the effects of concurrency are barely visible: aggregate throughput is stable up to 128 concurrent backups, but at 256 concurrent streams the overhead of pthread shared locks used for protected accesses to the FP cache buckets, as well as a few cache evictions that render the cache less "warm", take their toll—slightly lowering the aggregate throughput (10.6 GB/sec).

SSD indexing throughput. Using SSD index implementation on an 128 GB SSD drive, we repeated the throughput experiments of Figure 4 in order to 1) test the efficiency of our SSD indexing design, and 2) verify the effects of shared locking to duplicate data backups—since the SSD index is read-only and uses no shared locks. For our tests, we maintained the same sampling rate ($R = 1/101$) and used the same amount of memory for caching as before (2 GB)—so as to make a fair comparison. Notice that with this setup we are now using a total of only 10 GB and the amount of raw storage the system can support rose from 500 to 1,600 TB. Due to our efficient SSD index design and the lack of shared locking, most throughput results were similar or superior to those of the memory index. Table 2 summarizes the results and difference between the SSD index and memory index throughput. Notice, however, that these results include the cost of updating the SSD every time 65,536 new sampled entries have accumulated. A less (more) frequent SSD update policy would yield faster (slower) throughput results.

CPU cores	Unique data	100% Duplicates (cold cache)	100% Duplicates (warm cache)
1	347	354	356
2	599	612	612
4	900	1,167	1,172
8	907	1,983	2,004
14	925	2,373	2,485

Table 3: End-to-end backup throughput using a varying number of CPU cores. All numbers in MB/sec.

End-to-end throughput. Our next test attempted to include client performance in our evaluation, in an end-to-end system test, using a single 25 GB backup stream of unique segments. As presented in Table 3, we varied the number of CPU cores dedicated to MD5 calculation, and performed three tests for each configuration: an initial backup, a second backup of the same data with cold caches, and a third run with warm caches. All backups were performed using a 16-core Intel Xeon E5520 "client", with 32 GB of RAM, running RedHat Enterprise Linux 5. The results of Table 3 show that backing up unique data does not get much faster with more than 4 cores. Careful observation revealed two reasons for this behavior. First, even when using the Linux loopback interface, we could not get throughput higher than 10 Gbps, on that particular host. Notice that when bulk data transfers become unnecessary, the performance reaches 2.49 GB/sec. Second, we realized that careful optimization of our simple RPC mechanism might be able to yield better performance. However, optimizing network behavior and the RPC implementation is beyond the scope of this study. In order to evaluate the real throughput of our client design we made the assumption of an infinitely fast network/RPC infrastructure, and, temporarily, eliminated the network performance bottleneck. This revealed the client's full potential: running on our (slower) main Intel Xeon 5450 server, the client was able to push 360/697/1,023/1,319 MB/sec of unique data, with 1/2/3/4 cores dedicated to MD5, respectively.

Backup throughput conclusions. In summary, our backup throughput experiments show that, when backing up unique data, our system is nearly as efficient as a normal file server for single stream backup (no penalty for deduplication) and several times faster for multi-stream backups. This shows that our system can better organize the data on disks to achieve high throughput even with concurrent backups. When data are mostly duplicates, we can achieve 950 MB/sec for single stream backup and 6 GB/sec for multi-stream backups. Multiple

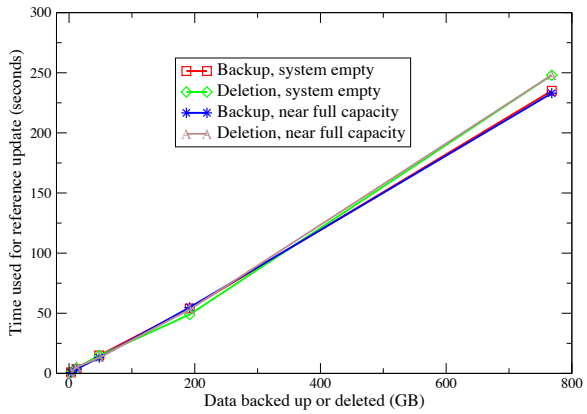


Figure 5: The reference update time for a given amount of data backed up or deleted when the system is empty and nearly full. The time is proportional to the data changed, and the slope shows the update throughput (3 GB/sec). Notice that the throughput is stable regardless of the capacity of the system or the amount of changed data.

streams help improve the aggregate throughput because they maximize the throughput of container FP catalog pre-fetching.

The major limitations that we observed are due to hardware restrictions: limited container pre-fetching throughput and CPU/networking bottlenecks in our end-to-end performance tests. On a production system equipped with hundreds of fast-seeking physical disks, and utilizing faster network connectivity, we expect to see much higher throughput. The only software limitation we observed was due to pthread locks, and is considered of secondary importance since it only impacts throughput minimally for more than 128 concurrent backup streams.

4.1.2 Reference Update Throughput

A critical property that is not often tested in deduplication systems, is the performance of reference updates, especially when we need to delete data—an operation that happens almost daily. Figure 5 shows reference update times measured when the synthetic data set gets backed up or deleted, both when the system is empty and near full capacity. The time is linear with the size of data backed up or deleted, because we need to update the reference of each segment that gets used.

The slope of the line corresponds to the throughput of the reference update, which is 3.2 GB/sec for data addition, and 3.1 GB/sec for data deletion. Deletion is slightly slower because when segments get deleted, they also need to be removed from the index. Contrary to a regular file-system, the deletion throughput of the deduplication system is slow because we pay the price of data sharing. However, it is still faster than the backup throughput of new data, which prevents the backup process from having to stall and wait for the deletion mechanism to free up space.

	Unique segs	Total unique	Ideal MBs	Real MBs	De-dupe
VM0	518,326	518,326	2,123	2,211	96%
VM1	733,267	921,522	3,775	3,938	96%
VM2	904,579	1,189,230	4,871	5,085	96%
VM3	1,145,029	1,616,585	6,621	6,860	97%

Table 4: Deduplication efficiency results for subsequent backups of four different versions of a Windows XP VMware image file.

4.1.3 Restore Throughput

Deduplication system benchmarks are dominated by backup testing and testing of restore is mostly ignored—probably because the restore process is usually slow, and correctness is the main concern. However, restore is an important operation and we wanted to ensure that our prototype provides sufficient performance. During our tests all data were restored correctly. Our single stream restore throughput was measured around 1 GB/sec, and 430 MB/sec for two or more concurrent restore streams. Single stream restore is fast because most accesses are sequential, while multiple concurrent restore streams introduce disk seeking. The use of directly locatable objects allows us to perform restore without using the index, making the whole process very scalable.

4.2 Deduplication Efficiency

Although we are willing to sacrifice some dedupe accuracy for high scalability, we still want to make sure the system provides adequate duplicate detection. In particular, since sampling provides the desired scalability, dedupe efficiency will be mostly determined by the effectiveness of pre-fetching.

In our synthetic data set, the true ("ground truth") duplication is 100%. Our prototype consistently eliminates no less than 97% of duplicates. This is consistent with the theoretical expectation, based on Equation 2: when we pre-fetch FPs from the container catalog, and because the sampling rate is 1 out of 101, the first 100 FPs may not be found. After the first hit, (101st FP in the worst case), we pre-fetch all FPs in that container. So theoretically we may fail to detect 100 over 4096 FPs, i.e., 2.4%.

For our VMWare data set we used our test sampling rate of 1/101, and a small FP cache (256 MB) in order to ensure that the cache cannot hold the whole working set. We performed multiple backups of each VM image, observing 100% dedupe efficiency for each run, with very high throughput (2.4 GB/sec). A more interesting experiment, however, presented in Table 4, is the dedupe efficiency achieved when backing up VM0, VM1, VM2, and VM3 back-to-back. Image VM0 has 518,326 4 KB segments, taking up 2,211 MB of disk space, instead

of 2,123 MB, giving us 96% of the ideal dedupe efficiency. Backing up VM1 introduced 403,196 new segments (330,071 of VM1's segments were also in VM0), taking up 3,938 MB, for a steady dedupe efficiency of 96%. Similarly, VM2 and VM3 were deduplicated at 96% and 97% of the optimal dedupe rate, which is a very satisfying result for a cache of only 256 MB. These results are particularly encouraging, since field experience has demonstrated that VM image backups are one of the most common and effective uses of dedupe.

4.3 Scalability

In order to test the scalability of the system we first populated it to near-full capacity (480 out of 500 TB i.e., 95.5%) with unique data. Because our disk array is only 24 TB, we stored everything except the actual segment data. As the code mainly operates on the metadata, discarding segment data has no impact on the correctness of the test. After the system was populated we repeated the same throughput tests, during which everything was stored on disk (including segment data).

Figure 6 presents a throughput comparison between an empty and near-full system. For multi-stream throughput, the system occupancy has negligible performance impact because for both unique and duplicate data the throughput is, once again, bounded by the disk's sequential write and random read performance, respectively. When the system is near full capacity, the index lookup and update time increase slightly. But the main bottleneck is still disk I/O—overshadowing the effects of CPU overhead. This means that the throughput of the system will scale well in terms of system capacity while disk I/O is the main bottleneck—which is probably going to be true in the foreseeable future.

The index overhead does show up for single stream throughput. The throughput of single stream backup near full capacity is slower than that of the empty system because single stream throughput is CPU bound and accessing a "fuller" index takes a little bit more CPU time.

Figure 5 also compares reference update performance when the system is empty and near-full. As expected, the time for reference update is almost the same, since the grouped mark-and-sweep algorithm only touches the changed backup groups. The majority of the references, regardless of how many they are when the system is near full capacity, are not touched by the grouped mark-and-sweep. Finally, we also checked the deduplication efficiency for both the synthetic and real data sets and observed no degradation in a near-full system.

Our results demonstrate that all parts of our prototype are able to scale to high capacity, with almost no performance decrease. We are confident that our system would scale to higher capacities, given more resources. Moreover, the raw capacity supported by our system (200 TB for every 10 GB of memory) is higher than the capacity of any other single-node system presented in Section 4.4.

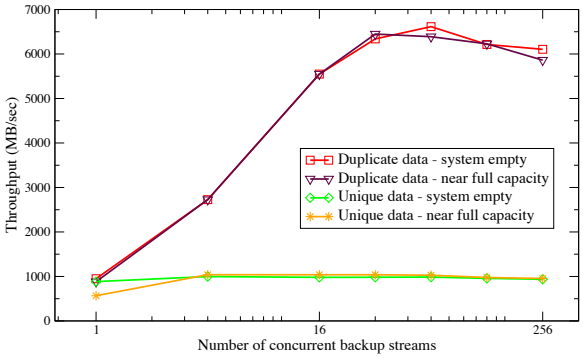


Figure 6: Throughput scalability tests show that there is no significant throughput drop when we get close to full capacity. We incur O(1) cost for most index operations, and throughput is disk-bound for both unique and duplicate data backups.

4.4 Comparison to State-of-the-art

When evaluating dedupe systems it is often the case that custom methods and private workloads are used to quantify the effectiveness of the proposed mechanisms (e.g., [19] and [12]). Comparisons to other systems are usually difficult, and limited to references to results reported by vendors, mostly because there is no agreed deduplication benchmark that would make benchmarking and comparisons fair and meaningful. Furthermore, when aiming to top the performance of state-of-the art systems, it is almost impossible to justify the cost and effort of obtaining, deploying and benchmarking even a single one of them. In our evaluation we tried to use data sets that will exercise the system in interesting ways, and that are relatively easy to be recreated and tested by other systems.

Table 4.4 presents some of the most popular high-performance deduplication solutions available as of April 2011. Assuming that all systems provide adequate deduplication efficiency (specifications do not provide precise numbers), we can see that our prototype's peak performance is similar to or better than that of all systems, with the exception of NEC's HydraStor. However, notice that HydraStor utilizes a large distributed system (55 "accelerator" and 110 "storage" nodes) in order to achieve its maximum throughput, and yet its raw capacity is limited to only 1.3 PB. Our prototype's single-node scalability competes with that of all systems and surpasses most of them, especially considering the limited amount of resources we have used (e.g., only 25 GB of RAM per 500 TB for $R = 1/101$, on an older 8-core server). Notice, however, that our goal to increase single-node scalability is not meant to replace all multi-node systems, but to potentially improve them by enabling each node to make better use of its resources and increase

Product	Backup (MB/sec)	Capacity (TB)	Nodes
DataDomain DD890 [3]	4,083	384	1
HP D2D4324 [7]	1,110	18	1
IBM ProtecTier [8]	1,000	1,000	2
Greenbytes GB4000 [6]	950	216	1
NEC HydraStor HS8-3110R [14]	41,250	1,300	55 + 110
Our prototype	6,000	500	1

Table 5: Summary: state-of-the art dedupe products as of April 2011.

data density per node. By doing so we could decrease the number of nodes necessary for a particular deployment, thus significantly decreasing the overall (acquisition, management, energy, etc) cost.

5 Related Work

Since the days of early deduplication systems, that performed mostly file-level or naive block-level deduplication [1, 11, 16], a lot of effort has been put into optimizing duplicate detection. In particular, many systems have investigated methods to perform content-aware segment boundary calculation, aiming to improve better duplicate coverage. Any degradation in dedupe efficiency was considered unacceptable. Such variable-size segmentation algorithms, utilize different variations of byte-level approaches, such as sliding window approaches (e.g., [5]), rolling hashes (e.g., [15]), Rabin fingerprints [2], and bimodal chunking [10]. For instance, systems like MAD2 [18], HYDRAstor [4, 17], as well as deduplication solutions by DataDomain [19] and Hewlett-Packard [12], utilize variable-size segments, in an attempt to achieve maximum compression. However, even if these algorithms make the best of raw storage (which is not always the case, as observed by [9]), single-node capacity is limited. Our work takes a different approach: we are willing to sacrifice some deduplication efficiency in order to achieve higher single-node scalability.

A sampling method is used in [12] to address indexing scalability restrictions. However, that approach is significantly different from ours, since it uses sampling to probabilistically identify "super-segments" that are used to perform coarse-granularity deduplication. Our segmentation algorithm operates at fine granularity at all times, and sampling is not used for pattern-matching, but for indexing actual file segments. Additionally, our approach is significantly more scalable, and can operate under heavy memory constraints, with good sampling rates: in a setting similar to the experiments presented in [12], our sampled index would require about 74% less memory (4.4 GB instead of 17 GB, with $R = 1/101$).

A lot of systems have used spacial locality to perform some type of caching (e.g., [18, 19]), but, to our knowledge, it has not been used before in combination with an aggressive sampling approach, such as the one we are proposing.

Our key assumption difference from previous efforts is that we are willing to relax our duplicate detection efficiency requirements, in order to address *all three* major challenges of single-node deduplication. Most other systems have provided good solutions for a subset of problems, usually excluding single-node scalability and reference management. For instance, DataDomain [19] addressed the disk bottleneck, by introducing a series of optimizations, including a Bloom filter, and spacial locality. However, their system can support a limited amount of raw storage, and is limited by network performance, since duplicate detection is performed only at the server. Additionally, it is not clear whether DataDomain's system can perform truly scalable resource reclamation.

HYDRAstor [4] on the other hand, achieves good scalability, but it does so by using a highly distributed, hierarchical model, with each node holding only a few tens of TB of storage. This design yields a high backup throughput, but at the cost of a highly distributed, costly system. Deletions in HYDRAstor, are implemented with a distributed reference counting method, which is difficult to maintain correctly, and scale without a large performance hit.

MAD2 [18] also uses a distributed storage system to provide scalability, as well as a number of optimizations that include spacial locality caching, and Bloom filters. Deletions are a very challenging operation in this system as well: they are performed only at the file level, and they are also handled by a variant of reference counting, with all the scalability and correctness problems discussed in Section 2.2. To our knowledge, our grouped mark-and-sweep approach is the only truly scalable, documented reference management implementation, that is also very resilient to errors.

Many scalable systems have adopted the event-driven design, however it is interesting that the nature of our application requires that we utilize it for the *client*, rather than the server. A pipelined client design was also proposed by [13], but it is significantly different from our design: it assumes pipeline stages whose operation requires a fixed amount of time, making it unrealistic for network operation. It also uses disk-based, client-side indexing, it implements a lot of functionality in the kernel, and it achieves scalability and throughput that is orders of magnitude lower than those of our client design.

6 Conclusion

Important engineering challenges need to be addressed in order to achieve the scalability, throughput and dedu-

plication efficiency necessary to provide next-generation deduplication support. We have presented a clean-slate design that aims to maximize overall single-node effectiveness, and introduces new mechanisms that address the most pressing of these challenges. Our directly locatable objects enable the use of progressive sampled indexing—in memory or on SSD—which provides superior single-node scalability and memory usage efficiency—unlike any other system we know of. Our grouped mark-and-sweep mechanism attacks the difficult, and often neglected, resource management and reclamation problem, in a truly scalable and efficient manner. Additionally, we have proposed an asynchronous interface to the server back-end, capable of pushing data to the server at a high-enough rate.

The performance of our prototype validates the effectiveness of our design. Progressive sampled indexing achieves very good deduplication efficiency, while using only 10 GB of memory per 200 TB of raw storage (25 GB for 500 TB in our tests). Additionally, we were able to achieve backup throughput ranging from 950 (all unique data) to 6,000 MB/sec (all duplicate data), with deduplication efficiency no less than 97%, while our grouped mark-and-sweep approach can process data with speeds higher than 3.1 GB/sec, demonstrating that single-node dedupe effectiveness can be greatly improved by making good use of available resources.

Acknowledgments

The authors would like to thank Weibao Wu, Joe Pasqua, Sanjay Sawhney, Kent Griffin, Tzi-cker Chiueh, Vish Janakiraman, Vic Pantaleon, and the Symantec PureDisk team for their input and their help on evaluating the system. We would also like to thank the anonymous reviewers for their valuable comments.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 1–14.

[2] BRODER, A. Z. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer-Verlag, pp. 143–152.

[3] DATADOMAIN. EMC data domain dd890. `http://bit.ly/eXL6Uc`.

[4] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. Hydrastor: a scalable secondary storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 197–210.

[5] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *KDD '05: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (New York, NY, USA, 2005), ACM, pp. 394–400.

[6] GREENBYTES. GB-X Series. `http://bit.ly/fTOSXd`.

[7] HP. StorageWorks D2D Backup Systems. `http://bit.ly/bfhyiU`.

[8] IBM. ProtecTIER Deduplication Solutions. `http://bit.ly/dVg3Z5`.

[9] JIN, K., AND MILLER, E. L. The effectiveness of deduplication on virtual machine disk images. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (New York, NY, USA, 2009), ACM, pp. 1–12.

[10] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *FAST'10: Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), USENIX Association, pp. 18–18.

[11] KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. Redundancy elimination within large collections of files. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2004), USENIX Association, pp. 5–5.

[12] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 111–123.

[13] LIU, C., XUE, Y., JU, D., AND WANG, D. A novel optimization method to improve de-duplication storage system performance. In *ICPADS '09: Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 228–235.

[14] NEC. HYDRAstor HS8-3000 Series. `http://bit.ly/hCHxhn`.

[15] NETAPP. Deduplicating Backup Data Streams with the NetApp VTL, 2009. `http://bit.ly/buXcaV`.

[16] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies* (Berkeley, CA, USA, 2002), USENIX Association, pp. 89–101.

[17] UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CALKOWSKI, G., DUBNICKI, C., AND BOHRA, A. Hydrafs: a high-throughput file system for the hydrastor content-addressable storage system. In *FAST'10: Proceedings of the 8th USENIX conference on File and storage technologies* (Berkeley, CA, USA, 2010), USENIX Association, pp. 17–17.

[18] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *MSST'10: Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies* (2010), pp. 1–14.

[19] ZHU, B., LI, K., AND PATTERSON, R. H. Avoiding the disk bottleneck in the datadomain deduplication file system. In *FAST '08: Proceedings of the Conference on File and Storage Technologies* (2008), pp. 269–282.

SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput

Wen Xia [†]
wx.hust@gmail.com

Hong Jiang [‡]
jiang@cse.unl.edu

Dan Feng [†] ✉
dfeng@hust.edu.cn

Yu Hua [†,‡]
csyhua@hust.edu.cn

[†] School of Computer, Huazhong University of Science and Technology, Wuhan, China
Wuhan National Lab for Optoelectronics, Wuhan, China
[‡]Dept. of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE, USA

Abstract

Data Deduplication is becoming increasingly popular in storage systems as a space-efficient approach to data backup and archiving. Most existing state-of-the-art deduplication methods are either locality based or similarity based, which, according to our analysis, do not work adequately in many situations. While the former produces poor deduplication throughput when there is little or no locality in datasets, the latter can fail to identify and thus remove significant amounts of redundant data when there is a lack of similarity among files. In this paper, we present SiLo, a near-exact deduplication system that effectively and complementarily exploits similarity and locality to achieve high duplicate elimination and throughput at extremely low RAM overheads. The main idea behind SiLo is to expose and exploit more similarity by grouping strongly correlated small files into a segment and segmenting large files, and to leverage locality in the backup stream by grouping contiguous segments into blocks to capture similar and duplicate data missed by the probabilistic similarity detection. By judiciously enhancing similarity through the exploitation of locality and vice versa, the SiLo approach is able to significantly reduce RAM usage for index-lookup and maintain a very high deduplication throughput. Our experimental evaluation of SiLo based on real-world datasets shows that the SiLo system consistently and significantly outperforms two existing state-of-the-art system, one based on similarity and the other based on locality, under various workload conditions.

1 Introduction

As the amount of the important data that needs to be digitally stored grows explosively to a worldwide storage crisis, data deduplication, a space-efficient method, has gained increasing attention and popularity in data storage. It splits files into multiple chunks that are each uniquely identified by a 20-byte SHA-1 hash signature, also called a fingerprint [21]. It removes duplicate chunks by checking their fingerprints, which avoids a byte-by-byte comparison. Data deduplication not only reduces the storage space overheads, but also minimizes the network transmission of redundant data in the network storage system [19].

One of the main challenges for centralized backup services based on deduplication is the scalability of fingerprint-index search. For example, to backup a dataset of 800TB and assuming an average chunk size of 8KB, at least 2TB of fingerprints have to be generated, which will be too large to be stored in the memory. Since the access to on-disk index is at least 1000 times slower than that to RAM, the frequent accesses to on-disk fingerprints are not acceptable for backup services and have become the main performance bottleneck of such deduplication systems.

Most of the existing solutions aim to make the full use of RAM, by putting the hot fingerprints into RAM to minimize accesses to on-disk index and improve the throughput of deduplication. There are two primary approaches to scaling data deduplication: locality based acceleration of deduplication, and similarity based deduplication. Locality-based approaches exploit the inherent locality in a backup stream, which is widely used in state-of-the-art deduplication systems such as DDFS [26] and ChunkStash [8]. Locality in this context means that the chunks of a backup stream will appear in approximately the same order in each full backup with a high probability. Exploitation of this locality increases the RAM utilization and reduces the accesses to on-disk index, thus alleviating the disk bottleneck. Similarity-based approaches are designed to address the problem encountered by locality-based approaches in backup streams that either lack or have very weak locality (e.g., incremental backups). They exploit data similarity instead of locality in a backup stream, and reduce the RAM usage by extracting similar characteristics from the backup

stream. A well-known similarity-based approach is Extreme Binning [3] that exploits the file similarity to achieve a single on-disk index access for chunk lookup per file.

While these scaling approaches have significantly alleviated the disk bottleneck in data deduplication, there are still substantial limitations that prevent them from reaching the peta- or exa-scale, as explained below. Based on our analysis of experimental results, we find that in general a locality-based deduplication approach performs very poorly when the backup stream lacks locality while a similarity-based approach underperforms for a backup stream with a weak similarity. Unfortunately, the backup data in practice are quite complicated in how or whether locality/similarity is exhibited. In fact, DDFS is shown to run very slowly in backup streams with little or no locality (e.g., when users only do the incremental backup). On the other hand, the similarity-based Extreme Binning approach is shown to fail to find significant amount of duplicate data in datasets with little or no file similarity (e.g., when the files are edited frequently). Fortunately, our preliminary study indicates that the judicious exploitation of locality can compensate for the lack of similarity in datasets, and vice versa. In other words, both locality and similarity can be complementary to each other, and can be jointly exploited to improve the overall performance of deduplication.

To this end, we propose SiLo, a scalable and low-overhead near-exact deduplication system, to overcome the aforementioned shortcomings of existing state-of-the-art schemes. The main idea of SiLo is to consider both similarity and locality in the backup stream simultaneously. Specifically, we expose and exploit more similarity by grouping strongly correlated small files into a segment and segmenting large files, and leverage locality in the backup stream by grouping contiguous segments into blocks to capture similar and duplicate data missed by the probabilistic similarity detection. The main contributions of this paper include:

- SiLo proposes a new similarity algorithm that groups many small strongly-correlated files into a segment or segments a large file to better expose and exploit their similarity characteristics. This grouping of small files results in much smaller similarity index for segments than chunk index, which can easily fit into RAM for a much larger dataset. The segmenting of large files can expose and thus extract more similarity characteristics so as to remove duplicate data with a higher probability.

- SiLo proposes an effective approach to mining the locality characteristics to capture similar and duplicate data missed by the probabilistic similarity detection by grouping multiple contiguous segments into a block, the basic cache and write-buffer unit, while preserving the spatial locality inherent in the backup stream on the disk. By keeping the similarity index and preserving spatial locality of backup streams in RAM (i.e., hash table and locality cache), SiLo is able to remove large amounts of redundant data, dramatically reduce the numbers of accesses to on-disk index, and substantially increase the RAM utilization.

- Our experimental evaluation of SiLo, based on real-world datasets, shows that the SiLo system consistently and significantly outperforms two existing state-of-the-art systems, the similarity-based Extreme Binning system and the locality-based ChunkStash system, under various workload conditions. According to our evaluations on duplicate elimination, SiLo can remove 1%$\sim$28% more redundant data than Extreme Binning and only 0.1%$\sim$1% less than the exact-deduplicating ChunkStash. Our evaluations on deduplication throughput (MB/sec) suggest that SiLo outperforms ChunkStash by a factor of about 3 and Extreme Binning by a factor of about 1.5. On the RAM utilization for the same datasets, SiLo consumes a RAM capacity that is only 1/41$\sim$1/60 and 1/3$\sim$1/90 respectively of that consumed by ChunkStash and Extreme Binning.

The rest of the paper is organized as follow. Section 2 presents background and motivation for this research. Section 3 describes the architecture and the design of the SiLo system. Section 4 presents our experimental evaluation of SiLo and discusses the results, including the comparisons with the state-of-the-art ChunkStash and Extreme Binning systems. Section 5 gives an overview of related work, and Section 6 draws conclusions and outlines future work.

2 Background and Motivation

In this section, we first provide the necessary background for our SiLo research by introducing the existing acceleration approaches for data deduplication, and then motivate our research by analyzing our observations based on extensive experiments on locality- and similarity-based deduplication acceleration approaches under real-word workloads.

2.1 Deduplication Acceleration Approaches

Previous studies have shown that the main challenge facing data deduplication lies in the on-disk index-lookup

bottleneck [26, 15, 8]. As the size of dataset to be deduplicated increases, so does the total size of fingerprints required to detect duplicate chunks, which can quickly overflow the RAM capacity for even high TB-scale and low PB-scale datasets. This can result in frequent disk accesses for fingerprint-index lookups, thus severely limiting the throughput of the deduplication system. Currently, there are two general approaches to accelerating the index-lookup of deduplication and alleviating the disk bottleneck, namely, the locality based and the similarity based methods.

The locality in the former refers to the observation that files, say A and B (thus their data chunks), in a backup stream appear in approximately the same order throughout multiple full backups with a high probability. DDFS [26] makes full use of this locality characteristic by storing the chunks in the order of the backup stream on the disk and preserving the locality in the RAM. It significantly reduces accesses to the on-disk index by increasing the hit ratio in the RAM. It also uses Bloom filters to quickly identify new (non-duplicate) chunks, which helps compensate for the cases where there is no or little locality, but at the cost of significant RAM overhead. Sparse Indexing [15] improves this method by sampling index instead of using Bloom filters. It uses less than half of the RAM capacity of DDFS. As a novel content-defined chunking algorithm, Bimodal [14] suggests that the neighboring data of duplicate chunks should be assumed to be good deduplication candidates due to backup-stream locality, which can be exploited to maximize the chunk size.

Nevertheless, all these approaches still produce unacceptable performance in face of very large datasets with little or no locality. The similarity-based approaches are proposed to exploit the similar characteristics in backup streams to minimize the chunk-lookup index in the memory. For example, Extreme Binning [3] exploits the similarity among files instead of locality, allowing it to make only one disk access for chunk lookup per file. It significantly reduces the RAM usage by storing only the similarity-based index in the memory. But it often fails to find significant amounts of redundant data when similarity among files is either lacking or weak. It puts similar files in a bin whose size grows with the size of the data, resulting in decreased throughput as the size of the similarity bin increases.

2.2 Small Files and Large Files

Our experimental observations, as well as intuition, suggest that the deduplication of small files can be very space and time consuming. A file system typically contains a very large number of small files [1]. Since the small files (e.g., $\leq$64KB) usually only take up a small fraction (e.g., $\leq$20%) of the total space of a file system but account for a large percentage (e.g., $\geq$80%) of the number of files, the chunk-lookup index for small files will be disproportionally large and likely out of memory. Consequently, the inline deduplication [26, 15] of small files will tend to be very slow and inefficient because of the more frequent accesses to the on-disk index for chunk lookup and the higher network-protocol costs between the client and the server.

This problem of small files can be addressed by grouping many highly correlated small files into a segment. We consider files with the logic sequence within the same parent directory to be highly correlated and thus similar. We exploit the similarity and the locality of a group (i.e., segment) of adjacent small files rather than one individual file or chunk. As a result, at most one access to on-disk index is needed per segment instead of per file or per chunk. The segmenting approach can also minimize the network costs by avoiding the frequent inline interactions per file.

A typical file system also contains many large files (e.g., $\geq$2MB) that only account for a small fraction (e.g., $\leq$20%) of total number of files but occupy a very large percentage (e.g., $\geq$ 80%) of the total space [1]. Obviously, these large files are an important consideration for a deduplication system due to their high space-capacity and bandwidth/time requirements in the backup process. When a large file is being deduplicated inline, the server must often wait for a long time for the chunking and hashing processes, resulting in low efficiency of the deduplication pipeline. In addition, the larger the files, the less similar they will appear to be even if significant parts within the files may be similar or identical, which can cause the similarity-based approaches to miss the identification of significant redundant data in large files.

To address this problem of large files, our SiLo approach divides a large file into many small segments to better expose similarity among large files while increasing the efficiency of the deduplication pipeline. More specifically, the probability that file S_1 and file S_2 share the same representative fingerprint is highly dependent on their similarity degree according to Broder's theorem [5]:

Theorem 1: Consider two sets S_1 and S_2, with H(S_1) and H(S_2) being the corresponding sets of the hashes of the elements of S_1 and S_2 respectively, where H is chosen uniformly and randomly from a min-wise independent family of permutations. Let min(S) denote the smallest element of the set of integers S. Then:

$$Pr[min(H(S_1) = min(H(S_2)))] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

This probability can be increased by segmenting the files and detecting all the segments of the file, as follows:

$$\Pr[\min(H(S_1) = \min(H(S_2)))] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \ll$$
$$\Pr[\min(H(S_{11}) = \min(H(S_{21})))] \cup \cdots \cup \Pr[\min(H(S_{1n}) = \min(H(S_{2n})))]$$
$$= \bigcup_{i=1}^{n} \Pr[\min(H(S_{1i}) = \min(H(S_{2i})))]$$
$$= 1 - \bigcap_{i=1}^{n} \Pr[\min(H(S_{1i}) \neq \min(H(S_{2i})))]$$
$$= 1 - \prod_{i=1}^{n} (1 - \frac{|S_{1i} \cap S_{2i}|}{|S_{1i} \cup S_{2i}|})$$

As files S_1 and S_2 are segmented into $S_{11} \sim S_{1n}$ and $S_{21} \sim S_{2n}$ respectively, the detection of similarity between S_1 and S_2 is determined by the union of the probabilities of detections of similarity between $S_{11} \sim S_{1n}$ and $S_{21} \sim S_{2n}$. Based on the above probability analysis, this segmenting approach will only fail in the worst-case scenario where all the segments in file S_1 are not similar to segments of file S_2. This, based on the inherent locality in the backup streams, happens with a very small probability because it is extremely unlikely that two files are very similar but none or very few of their respective segments are detected as being similar.

2.3 Similarity and Locality

Now we further analyze the relationship between similarity and locality with respect to backup streams. As mentioned earlier, chunk locality can be exploited to store and prefetch groups of contiguous chunks that are likely to be accessed together with a high probability in the backup stream, while files' similarity may be mined so that the similarity characteristics instead of the whole sets of fingerprints of files, are indexed to minimize the index size in the memory. The exploitation of locality maximizes the RAM utilization to improve the throughput but can cause RAM overflows and frequent accesses to on-disk index when datasets lack or are weak in locality.

The similarity-based approaches minimize the RAM usage at the cost of potentially missing large amounts of redundant datawhich is dependent on the similarity degree of the backup stream. We have examined the similarity degree and the duplicate-elimination measure of our similarity-only deduplication approach on four datasets, as shown in Figure 1 and Figure 2. The four datasets represent one-backups, incremental-backups, Linux-versions and full-backups respectively, whose characteristics will be detailed in Section 4.

The similarity degree is computed by our similarity detection on the Linux dataset as: Simi(S_{input}) = Max (| $S_{input} \cap S_i$ |)/$|S_{input}|$,($S_i \in S_{store}$, Simi(S_{input}) $\in$ [0,1]). Thus, the similarity degree "1" signifies that the input segment is completely duplicate and the similarity degree "0" states that the segment is detected to match no other segments at all by our similarity-only approach.

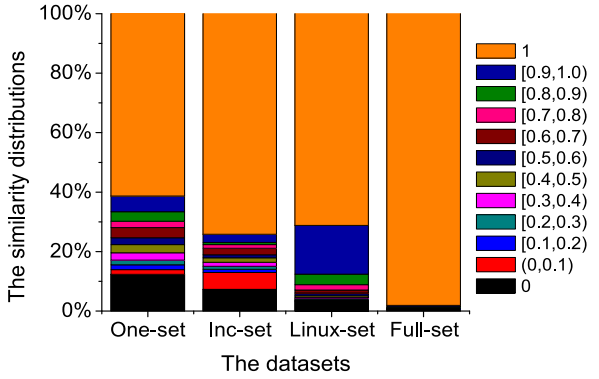


Figure 1: The distribution of the segment similarity on four datasets by our similarity-only approach. It can be used to describe the similarity characteristics of datasets. A large proportion of data with low similarity degrees is observed here.

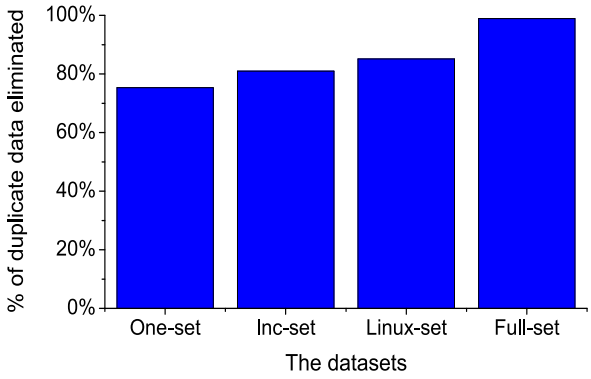


Figure 2: Percentage of duplicate data eliminated by our similarity-only deduplication approach.

Figure 3 further examines the duplicate elimination missed by the similarity approach on the Linux dataset. The missed portion of duplication elimination is defined as the difference between the measure achieved by the exact deduplication and that by the similarity-based deduplication. Therefore, Figure 3 shows that the similarity-based deduplication efficiency is heavily dependent on the similarity degree of the backup stream which is well consistent with Broder's Theorem in (see Section 2.2). The similarity approach often fails to remove large amounts of duplicate data, especially when the backup stream has a low similarity degree.

Inspired by Bimodal [14], which shows that the backup-stream locality can be mined to find more potentially duplicate data, we believe that such locality can also be mined to expose and thus detect more data similarity, a point well demonstrated by our experimental study in Section 4. More specifically, SiLo mines lo-

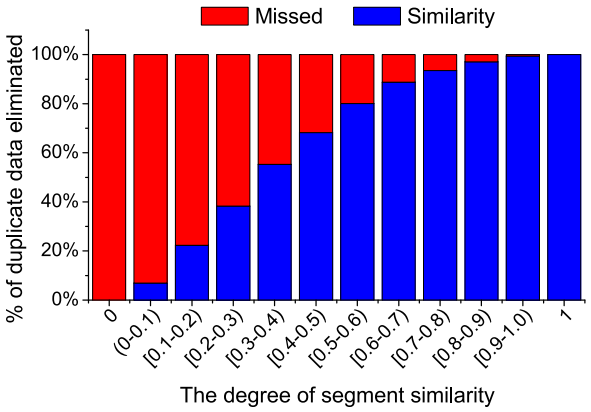


Figure 3: Percentage of duplicate data eliminated as a function of different similarity degree on the Linux dataset by our similarity-only deduplication approach.

cality in conjunction with similarity by grouping multiple contiguous segments in a backup stream into a block. While this exploitation of locality helps find more potential deduplication candidates by detecting similar segments' adjacent segments in a block, it also reduces the accesses to on-disk index to improve the deduplication throughput.

Now we analyze the combined exploitation of similarity and locality. Given two blocks B_1 and B_2, each containing n segments ($S_{11} \sim S_{1n}$, $S_{21} \sim S_{2n}$), according to the Broder's theorem, the percentage of duplicate eliminated by the similarity-only approach can be computed as: $DeDup_{Simi}(B_1, B_2) = |B_1 \cap B_2| \ / \ |B_1 \cup B_2|$. The combined and complementary exploitation of similarity and locality can be computed as follows:

$$
\begin{aligned}
& DeDup_{SiLo}(B_1, B_2) \\
&= \bigcup_{i=1}^{n} Pr[min(H(S_{1i}) = min(H(S_{2i})))] \\
&= 1 - \bigcap_{i=1}^{n} Pr[min(H(S_{1i}) \neq min(H(S_{2i})))] \\
&= 1 - \prod_{i=1}^{n} (1 - \frac{|S_{1i} \cap S_{2i}|}{|S_{1i} \cup S_{2i}|}) \\
&= 1 - (1-a)^N (assume\ all\ the\ \frac{|S_{1i} \cap S_{2i}|}{|S_{1i} \cup S_{2i}|} = a)
\end{aligned}
$$

Assume that the value a follows a uniform distribution in the range [0,1] (It may be much more complicated in the real world datasets), the expected value of duplicate elimination can be further calculated under the aforementioned assumption as:

$$
\begin{aligned}
E_{Simi} &= \int_0^1 (a) da = \frac{1}{2} \\
E_{SiLo} &= \int_0^1 (1 - (1-a)^N) da = \frac{N}{N+1}
\end{aligned}
$$

Thus the larger the value N (i.e., the number of segments in a block), the more locality can be exploited in deduplication. E_{Simi} is equal to E_{SiLo} when N=1. SiLo can remove more than 99% of duplicate data when N $>$ 99. Thus the combined exploitation of similarity and locality makes it possible to achieve the near-complete duplicate elimination (recall that exact deduplication achieves complete duplicate elimination) and requires at most one disk access per segment (a group of chunks or small files) rather than one access per chunk (as in locality-based approaches) or per file (as in similarity-based approaches), thus avoiding the disk bottleneck of data deduplication. In addition, the throughput of the deduplication system also tends to be improved by reducing the expensive accesses to on-disk index. As a result, our SiLo approach, through its judicious and joint exploitation of locality and similarity, is able to significantly improve the overall performance of the deduplication system as demonstrated in Section 4.

3 Design and Implementation

SiLo is designed for large-scale and disk-inline backup storage systems. In this section, we will first describe the architecture of SiLo, followed by detailed discussions of its design and implementation issues.

3.1 System Architecture Overview

As depicted in Figure 4, the SiLo architecture consists of four key functional components, namely, File Deamon (FD), Deduplication Server (DS), Storage Server (SS), and Backup Server (BS), which are distributed in the datacenters to serve the backup requests. BS and DS reside in the metadata server (MDS) while FD is installed on each client machine that requires backup/restore services.

- File Deamon is a deamon program providing a functional interface (e.g., backup/restore) in users' computers. It is responsible for gathering backup datasets and sending/restoring them to/from Storage Servers for backups/restores. The processes of chunking, fingerprinting and segmenting can be done by FD in the preliminary phase of the inline deduplication. It also includes a File Agent that is responsible for communicating with BS and DS and transferring backup data to/from SS.

- Backup Server is the manager of the backup system that globally manages all jobs of backup/restore and directs all File Agents and Storage Servers. It maintains a metadata database for administering all backup files' information.

- The main function of Deduplication Server is to store and look up all fingerprints of files and chunks.

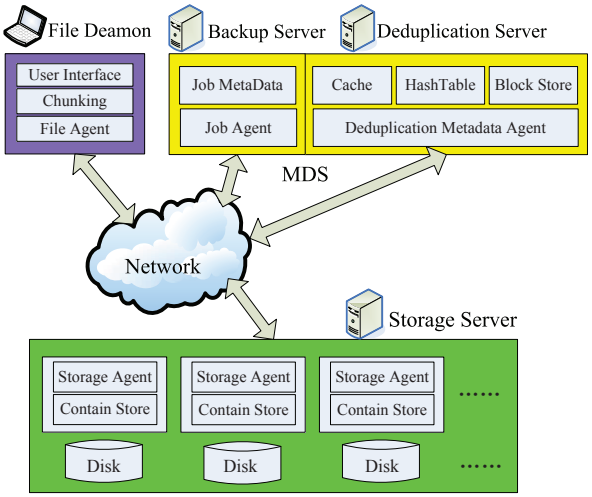


Figure 4: The SiLo system architecture.

- Storage Server is the repository for backed-up data. SS in SiLo manages multiple Storage Nodes for scalability and provides fast, reliable and safe backup/restore services.

In this paper, we focus on Deduplication Server since it is the most likely performance bottleneck of the entire deduplication system. DS consists of the locality hash table (LHTable), the similarity hash table (SHTable), write buffer and read cache. While SHTable and LHTable index segments and blocks, the similarity and locality units of SiLo respectively, the write buffer and read cache preserve the similarity and locality of the backup stream, as shown in Figure 5.

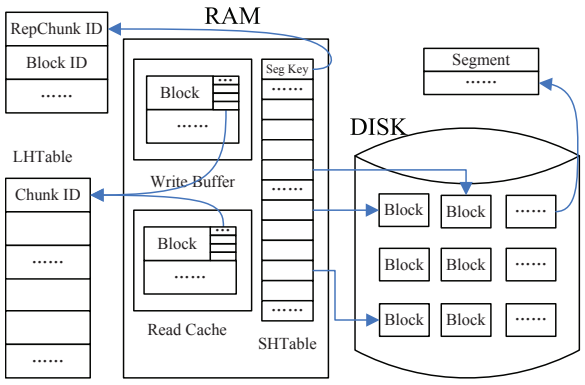


Figure 5: Data structures of Deduplication Server.

The notion of segment is used to exploit the similarity of the backup stream while the block preserves the stream-informed locality layout of segments on the disk. SHTable provides the similarity detection for input segments and LHTable serves to quickly index and filter out duplicate chunks. Note that, since this paper mainly aims at improving the performance of accessing on-disk fingerprints in the deduplication system, all write/read operations in this paper are performed in the form of writing/reading chunks' fingerprints rather than the real backup data.

3.2 Similarity Algorithm

As mentioned in Section 2.3, SiLo exploits similarity and locality jointly. It exploits similarity by grouping strongly correlated small files and segmenting large files, while locality is exploited by grouping contiguous segments in a backup stream to preserve the locality layout of these segments as depicted in Figure 6. Thus, segments are the atomic building units of a block that is in turn the atomic unit of the write buffer and the read cache.

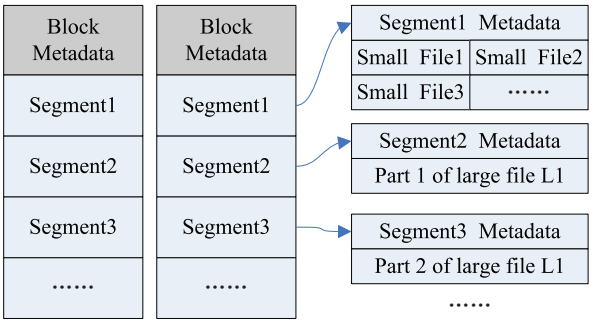


Figure 6: Data structure of the SiLo similarity algorithm.

As a salient feature of SiLo, the SiLo similarity algorithm is implemented in File Deamon, which structures data from backup streams into segments according to the following three principles.

- P1. Correlated small files in a backup stream (e.g., those under the same parent directory) are to be grouped into a segment.
- P2. A large file in a backup stream is divided into several independent segments.
- P3. All segments are of approximately the same size (e.g., 2MB).

Where, P1 aims to reduce the RAM overhead of index-lookup; P2 helps expose more similarity characteristics of large files to eliminate more duplicate data; and P3 simplifies the management of segments. Thus, the similarity algorithm exposes and then exploits more similarity by leveraging file semantics and preserving locality-layout of a backup stream to significantly reduce the RAM usage.

SiLo employs the method of representative fingerprinting [3] to represent each segment by a similarity-index entry in the similarity hash table. By virtue of P1, the SiLo similarity design solves the problem of small

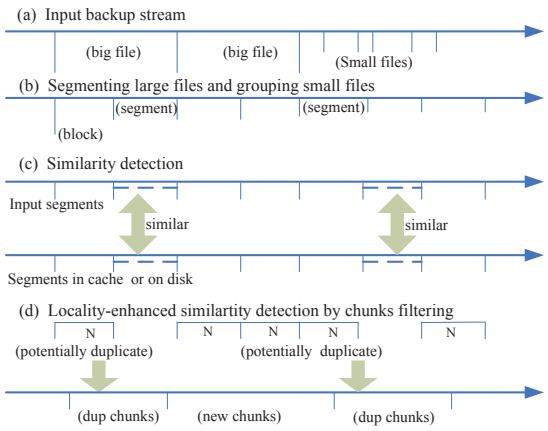


Figure 7: The workflow of the locality algorithm: it helps detect more potentially duplicate chunks that are missed by the similarity algorithm.

files taking up disproportionally large RAM space. For example, assuming an average segment size of 2MB and an average chunk or small file size of 8KB, a segment accommodates 250 chunks or small files, thus significantly reducing the required index size in the memory. If we assume a 60-byte primary key for the similarity indexing of a 2MB segment of backup data, which is considered economic, a 1TB backup stream only needs 30MB similarity-index for deduplication that can easily fit in the memory.

3.3 Locality Algorithm

As another salient feature of SiLo, the SiLo locality algorithm groups several contiguous segments in a backup stream into a block and preserves their locality-layout on the disk. Since block is also the minimal write/read unit of the write buffer and read cache in the SiLo system, it serves to maximize the RAM utilization and reduce frequent accesses to on-disk index by retaining access locality in the backup stream. By exploiting the inherent locality in backup streams, the block-based SiLo locality algorithm is able to eliminate more duplicate data.

Figure 7 shows the workflow of the locality algorithm. According the locality characteristic of backup streams, if input segment S_{1k} in block B_1 is determined to be similar to segment S_{2k} by hitting in the similarity hash table, SiLo will consider the whole block B_1 to be similar to block B_2 that contains S_{2k}. As a result, this grouping of contiguous segments into a block can eliminate more potentially duplicate data that is missed by the probabilistic similarity detection, thus complementing the similarity detection.

When SiLo reads the blocks from disk by the similarity detection, it puts the recently accessed block into the read cache. By preserving the backup-stream locality in the read cache, the accesses to on-disk index due to similarity detection can be significantly reduced, which alleviates the disk bottleneck and increases the deduplication throughput. Since it is at the block level where locality is preserved and exploited, the block size is an important system parameter that affects the system performance such as duplicate elimination and throughput. The smaller the block size, the more disk accesses will be required by the server to read the index, weakening the locality exploitation. The larger the block size, on the other hand, the more unrelated segments will be read by the server from the disk, increasing system's space and time overheads. Therefore, a proper block size not only provides good duplicate elimination, but also achieves high throughput and low RAM usage in the SiLo system.

Each block in SiLo has its own Locality Hash Table (i.e., LHTable shown in Figure 5) for chunk filtering. Since a block contains several segments, it needs an indexing tool for thousands of fingerprints. The fingerprints in a block are organized into the LHTable when reading the block from the disk. The additional time required for constructing LHTable in a block is significantly compensated by its quick indexing.

3.4 Cache and RAM Considerations

SiLo uses a very small portion of RAM as its write buffer and read cache to store a small number of recently accessed blocks to avoid the frequent and expensive disk read/write operations. In our current design of SiLo, the read cache and the write buffer each contains a fixed number of blocks. As illustrated in Figures 5 and 6, a locality-block contains only metadata information such as LHTable, segment information, chunk information, and file information, which enables a 1MB locality-block to represent a 200MB data-block.

Since users of file systems tend to duplicate files or directories under the same directories, a significant amount of duplicate data can be eliminated by detecting the duplication in the write buffer that also preserves the locality of a backup stream. For example, a code directory may include many versions of source code files or documents that can be good deduplication candidates.

The largest portion of RAM in the SiLo system is occupied by the similarity hash table (i.e., SHTable shown in Figure 5). Assuming an average segment size of 2MB and a primary-key size of 60B, the SiLo SHTable requires 300MB for an average backup data of 10 TB. Thus the RAM usage for the cache becomes negligibly small as the data size further increases.

3.5 SiLo Workflow

To put things together and in perspective, Figure 8 shows the main workflow of the SiLo deduplication process. For an incoming backup stream, SiLo goes through the following key steps:

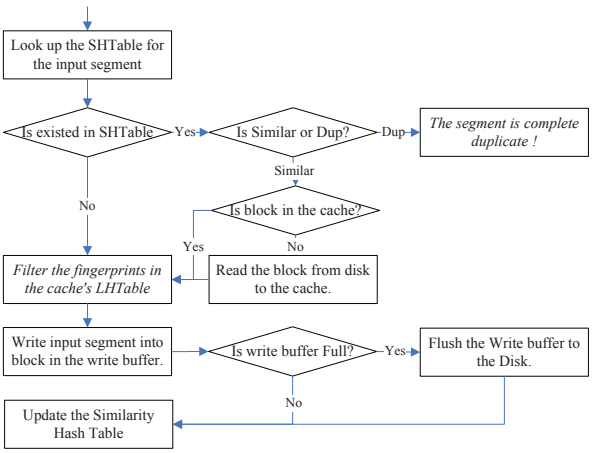


Figure 8: The SiLo deduplication workflow.

1. Files in the backup stream are first chunked, fingerprinted and packed into segments by grouping strongly correlated small files and segmenting large files in the File Agent.

2. Each newly generated segment S_{new} is checked against SHTable for similarity detection. If the new segment hits in SHTable, SiLo checks if the block B_{bk} containing S_{new}'s similar segment is in the cache. If it is not in the cache, B_{bk} is read from the disk to the read cache, where a block is replaced in the FIFO order if the cache is full. If S_{new} misses in SHTable, it is then checked against recently accessed blocks in the read cache for potentially similar segment in one of the cached blocks (B_{bk}).

3. The duplicate chunks in S_{new} are eliminated by checking LHTable of B_{bk} in the read cache. Then the chunks in the neighbouring segments of S_{new} in the backup stream are filtered by the locality-enhanced similarity detection (i.e., these chunks are checked against LHTable of B_{bk} for possible duplication).

4. After chunk filtering and constructing a new and non-duplicate block B_{new} from the backup stream, SiLo checks if the write buffer is full. If the write buffer is full, a block there is replaced in the FIFO order by B_{new} and then written to the disk.

As demonstrated in Section 4, SiLo is able to minimize both the time and space overheads of indexing fingerprints while maintaining a duplicate elimination performance comparable to exact deduplication methods such as ChunkStash.

4 Evaluation

In order to evaluate SiLo, we have implemented a prototype of SiLo that allows us to examine several important design parameters to provide useful insights. We compare SiLo with the similarity-based and locality-based state-of-the-art approaches Extreme Binning and ChunkStash in the key deduplication metrics of duplicate elimination, RAM usage and throughput. The evaluation is driven by four real-world traces collected from real backup datasets that represent different workload characteristics.

4.1 The Experimental Setup

We use a standard server configuration to evaluate and compare the inline deduplication performances of the SiLo, ChunkStash and Extreme Binning approaches running on a Linux environment. The hardware configuration includes a quad-core CPU running at 2.4GHz, with a 4GB RAM, 2 gigabit network interface cards, and two 500GB 7200rpm hard disks.

Due to our lack of access to the source code of either the ChunkStash or Extreme Binning scheme, we have chosen to implement both of them. More specifically, we have implemented the locality-based and exact-deduplication approach of ChunkStash incorporating the principles and algorithms described in the ChunkStash paper [8]. The ChunkStash approach makes full use of the inherent locality of backup streams and uses a novel data structure called Cuckoo hash for fingerprint indexing. We have also implemented a simple version of the Extreme Binning approach, which represents a similarity-based and approximate-deduplication approach according to the algorithms described in the Extreme Binning paper [3]. Extreme Binning exploits file similarity instead of locality in the backup streams.

Note that our evaluation platform is not a production-quality deduplication system but rather a research prototype. Hence, our evaluation results should be interpreted as an approximate and comparative assessment of the three systems above, and not be used for absolute comparisons with other deduplication systems. The RAM usage in our evaluation is obtained by recording the space overhead of index-lookup. The duplicate elimination performance metric is defined as the percentage of duplicate data eliminated by the system. Throughput of the system is measured by the rate at which fingerprints of the backup stream are processed, not the real

Feature	One-set	Inc-set	Linux	Full-set
Total size	530GB	251GB	101 GB	2.51TB
Total files	3.5M	0.59M	8.8M	11.3M
Total chunks	51.7M	29.4M	16.9M	417.6M
Avg.chunk size	10KB	8KB	5.9KB	6.5KB
Dedupe factor	1.7	2.7	19	25
Locality	weak	weak	strong	strong
Similarity	weak	strong	strong	strong

Table 1: Workload characteristics of the four traces used in the performance evaluation. All use SHA-1 for chunk fingerprints and the content-based chunking algorithm. The deduplication factor is defined as the Totalsize /(Totalsize - Dedupsize) ratio.

backup throughput in that it does not measure the rate at which the backup data is transferred and stored.

Four traces representing different strengths of locality and similarity are used in the performance evaluation of the three deduplication systems and are listed in Table 1. The four traces are collected from real-world datasets of One-backup, Incremental-backup, Linux-version and Full-backup respectively.

The One-set trace was collected from 15 graduate students of our research group. To obtain traces from this backup dataset, we have built a deduplication analysis tool that crawls the backup directory, and generates the sequences of chunk and file hashes for traces. Since we obtain only one full backup for this group, this trace has weak locality and weak similarity. The Inc-set is a subset of the trace reported by Tan et al. [24] and was collected from initial full backups and subsequent incremental backups of eight members in a research group. There are 391 backups with a total of 251GB data. Therefore, Inc-set represents datasets with strong similarity but weak locality.

Linux-set, downloaded from the website [16], consists of 900 versions from version 1.1.13 to 2.6.33, and represents the characteristics of small files. Full-set consists of 380 full backups of 19 researchers' PCs, which is also reported by Xing et al. [25] and can be downloaded from the website [10]. Full-set represents datasets with strong locality and strong similarity. Both Linux-set and Full-set are used in [25] and [3] to evaluate the performance of Extreme Binning, and our use of these datasets resulted in similar and consistent evaluation results with the published studies.

With the above traces representing different but typical workload characteristics, this evaluation intends to answer, among other things, the following questions: Can the SiLo locality algorithm compensate for the probabilistic similarity detection that may miss detecting large amounts of duplicate data? How effective is the SiLo similarity algorithm under different workload conditions? How is SiLo compared with existing state-of-the-art deduplication approaches in key performance measures?

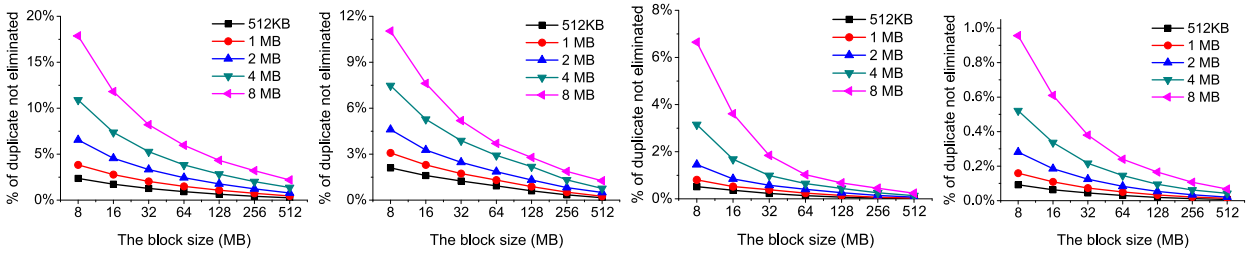
4.2 Interplay between Similarity and Locality

The mutually interactive nature of similarity and locality in SiLo dictates a good understanding of the relationship between locality and similarity before a thorough performance evaluation is carried out. Thus, we first examine the impact of the SiLo design parameters of block size and segment size on duplicate elimination and time overhead, which is critical for the SiLo locality and similarity algorithms.

From Figure 9 that shows the percentage of duplicate data not eliminated, we find that the duplicate elimination performance, defined as the percentage of duplicate data eliminated, increases with the block size but decreases with the segment size. This is because the smaller the segment is (e.g., segment size of 512KB), the more similarity can be exposed and detected, enabling more duplicate data to be removed. On the other hand, the larger the block is (e.g., block size of 512MB), the more locality of the backup stream will be retained and captured, allowing SiLo to eliminate more (i.e., 97%$\sim$99.9%) of redundant data regardless of the segment size.

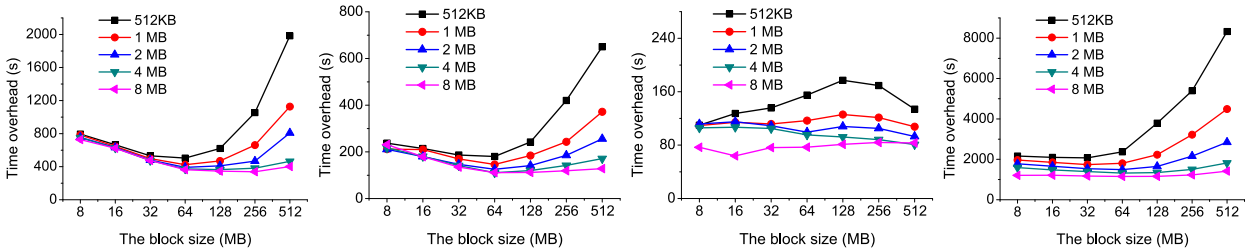
Although more redundant data can be eliminated by reducing the segment size or filling a block with more segmentsas indicated by the results shown in Figure 9, it results in more accesses to on-disks index and higher RAM usage due to the increased index entries in the SHTable (see Figure 5). As the deduplication-time-overhead results of Figure 10 clearly suggest, continuously decreasing the segment size or increasing the block size can become counterproductive after a certain point. From Figure 10, we further find that, for a fixed block size, the time overhead is inversely proportional to the segment size. This is consistent with our intuition that smaller segment size results in more frequent similarity detections for the input segments, which in turn can cause more accesses to on-disk index.

Figure 10 also shows that there is a knee point for each curve, meaning that for a given segment size and workload the time overhead decreases first and then increases (except Figure 10 (c)). This may be explained by the fact that, with a very small block size (e.g., 8MB), there is little locality to be mined, resulting in frequent accesses to on-disk index. With a very large block size (e.g., 512MB), SiLo also runs slower because the increased disk accesses for locality exploitation may result in more unrelated segments being read in. The Linux-set are different from other datasets in Figure 10, because the average size of a Linux version is 110MB, which enables



(a) One-set (b) Inc-set (c) Linux-set (d) Full-set

Figure 9: Percentage of duplicate data eliminated as a function of block size and segment size.



(a) One-set (b) Inc-set (c) Linux-set (d) Full-set

Figure 10: Time overhead of SiLo deduplication as a function of block size and segment size.

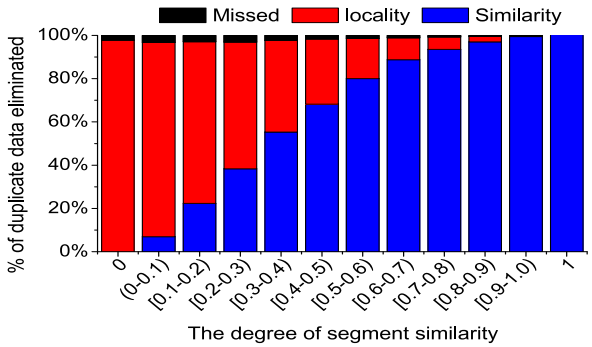


Figure 11: Percentage of duplicate data eliminated as a function of different similarity degrees on the Linux-dataset by the similarity-only approach and locality-only approach respectively.

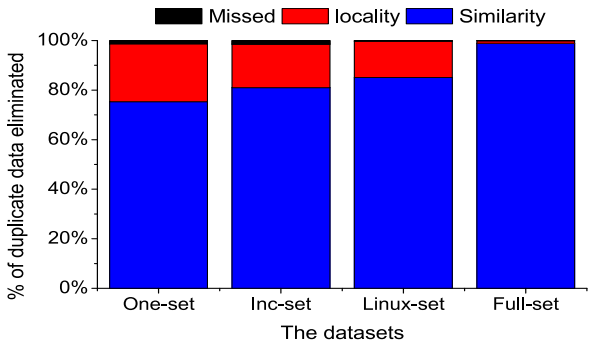


Figure 12: Percentage of duplicate data eliminated on four datasets by the similarity-only approach and locality-only approach respectively.

more related locality to be exploited at the block size of 256MB.

As analyzed above, there evidently exist an optimum segment size and an optimum block size, subject to a given workload and deduplication requirements (e.g., duplicate elimination or deduplication throughput). The choice of segment size and block size can be dynamically adjusted by the user's specific requirements (e.g., the backup throughput or duplicate elimination or the RAM usage).

Figures 11 and 12 suggest that the full exploitation of locality jointly with that of similarity can remove almost all redundant data missed by the similarity detection under all workloads. These results can be compared with Figure 2 and 3, then well verify our motivation of similarity and locality in Section 2. In fact, only an extremely small amount of duplicate data is missed by SiLo even on the datasets with weak locality and similarity.

4.3 Comparative Evaluation of SiLo

This subsection presents evaluation results comparing SiLo with two other state-of-the-art deduplication systems, the similarity-based Extreme Binning system and the locality-based ChunkStash system, by executing the four real-world traces described in Section 4.1 on these three systems. Note that in this evaluation SiLo assumes

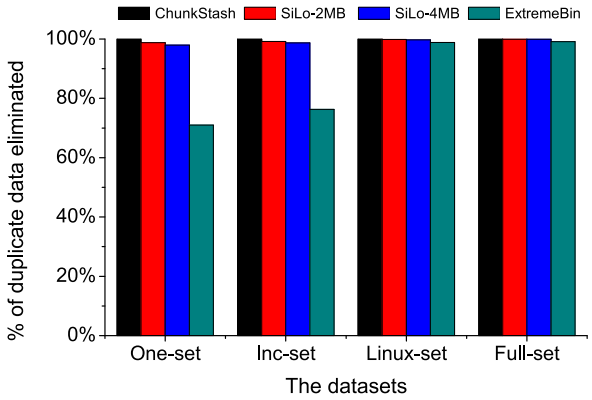


Figure 13: Comparison among ChunkStash, SiLo, and Extreme Binning in terms of percentage of duplicate data eliminated on the four datasets.

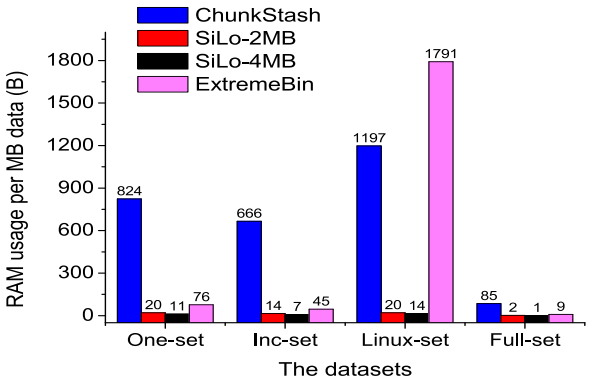


Figure 14: Comparison among ChunkStash, SiLo, and Extreme Binning in terms of RAM usage (B: RAM required per MB backup data).

a block size of 256MB, while SiLo-2MB and SiLo-4MB represent SiLo with a segment size of 2MB and 4MB respectively.

A. Duplicate elimination

Figure 13 shows the duplicate elimination performance of the three systems under the four workloads. Since ChunkStash does the exact deduplication, it eliminates 100% of duplicate data. Compared with Extreme Binning that eliminates 71%∼99% of duplicate data in the four datasets, SiLo removes about 98.5%∼99.9% of duplicate data. Note that, while Extreme Binning eliminates about 99% of duplicate data as expected in Linux-set and Full-set that has strong similarity and locality, it fails to detect almost 30% of duplicate data in One-set that has weak locality and similarity, and about 25% of duplicate data in Inc-set with weak locality but strong similarity. Although there is strong similarity in Inc-set, Extreme Binning still fails to eliminate a significant amount of duplicate data primarily due to its probabilistic similarity detection that simply chooses one representative fingerprint for each file regardless of the file size.

On the contrary, SiLo-2MB eliminates 99% of duplicate data even in One-set with both weak similarity and locality, and also removes almost 99.9% of duplicate data in Linux-set and Full-set with both strong similarity and locality. These results show that SiLo's joint and complementary exploitation of similarity and locality is very effective in detecting and eliminating duplicate data under all workloads evaluated, achieving near-complete duplicate elimination (i.e., exact deduplication).

B. RAM usage

Figure 14 shows the RAM usage for deduplication among these three systems under the four workloads. For Linux-set that has a very large number of small files and small chunks, the highest RAM usage is incurred for both Chunkstash and Extreme Binning. There is also a clear negative correlation between the deduplication factor and the RAM usage for the approximate deduplication systems of SiLo and Extreme Binning on the other four workloads. That is, for One-set that has the lowest deduplication factor, the highest RAM usage is incurred, while for Full-set that has highest deduplication factor, the smallest RAM space is required.

The average RAM usage for ChunkStash is the highest among the three approaches, except for the Linux-set trace, as it does the exact deduplication that needs a large hash table in the memory to put all the indices of chunk fingerprints. Although ChunkStash uses the Cuckoo hash to store compact key signatures instead of full chunk-fingerprints, it still requires at least 6 bytes for each new chunk, resulting in a very large cuckoo hash table for millions of fingerprints. In addition, according to the open-source code of Cuckoo Hash [14], the ChunkStash system needs to allocate about two million hash table slots in advance to support one million index entries.

Since only the file similarity index needs to be stored in RAM, Extreme Binning only consumes about 1/9∼1/15 of the RAM space required of ChunkStash except on the Linux-set where it consumes more RAM usage than ChunkStash due to the extremely large number of small files. However, SiLo-2MB's RAM efficiency allows it to reduce the RAM consumption of Extreme Binning by a factor of 3∼900. The extremely low RAM overhead of the SiLo system stems from the interplay between its similarity algorithm, which groups many small correlated files into segments and extracts their similarity characteristics, and its locality algorithm, which groups contiguous segments of the backup stream into blocks to effectively exploit the locality residing in the backup-streams. On the other hand, the RAM usage for Extreme Binning depends on the average file size of the file set, in addition to the deduplication factor. The smaller the average file size is, the more RAM space Extreme Binning will consume, which is demonstrated in the Linux-set.

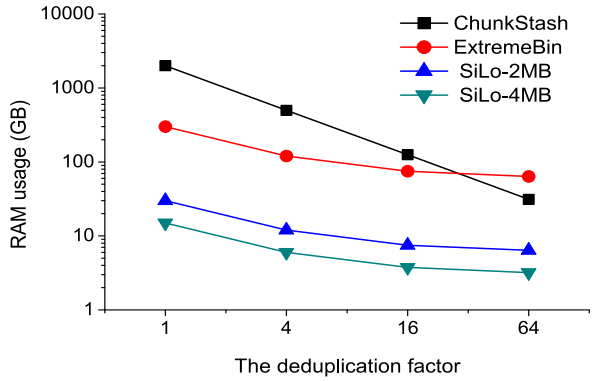


Figure 15: Comparison among ChunkStash, SiLo, and Extreme Binning in terms of RAM usage in PB-scale deduplication with different deduplication factors. We assume that the average file size is 200KB, and 80% of duplicate data are from duplicate files.

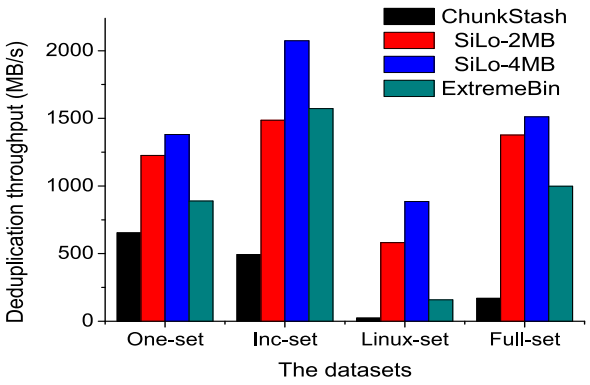


Figure 16: Comparison among ChunkStash, SiLo, and Extreme Binning in terms of deduplication throughput (MB/sec).

The RAM usage of the SiLo system remains relatively stable with the change in average file size in the four traces and is inversely proportional to the deduplication factor of the traces.

Now we analyze the RAM usage in a PB-scale deduplcation system for the three approaches. As a 2MB-segment needs 60 bytes of key index in the memory, SiLo takes up about 30GB of RAM in a PB-scale deduplication system. With 4MB-segments, SiLo's RAM usage is halved to 15 GB in a PB-scale deduplication system while its performance degrades gracefully as shown in Figures 9 and 10. Extreme Binning needs almost 300GB of RAM space with an average file size of 200KB while ChunkStash consumes almost 2TB of RAM space to maintain a global index in a PB-scale deduplication system.Figure 15 also shows RAM usage of these three approaches with different deduplication factors. According to [15], Sparse Indexing uses 170GB of RAM space for a PB-scale deduplication system, whereas it estimates that DDFS would require 360GB RAM to maintain a partial index depending on locality in backup streams.

C. Deduplication throughput

Figure 16 shows a comparison among the three approaches in terms of deduplication throughput, where the throughput is observed to more than double as the average chunk size changes from 6KB (e.g., Linux-set) to 10KB (e.g., One-set).

ChunkStash achieves an average throughput of about 335MB/sec with a range of 24MB/sec∼ 654MB/sec on the four datasets. The frequency of accesses to on-disk index by Chunkstash's compact key signatures algorithm on the Cuckoo hash lookup tends to increase with the size of the dataset, thus adversely affecting the throughput. Extreme Binning achieves an average throughput of 904MB/sec with a range of 158MB/sec∼1571/sec on the four datasets, since it only needs to access the disk once per similar-file and eliminates the duplicate files in the memory. As SiLo-2MB makes at most one disk access per segment, it deduplicates data at an average throughput of 1167 MB/sec with a range of 581MB/sec∼1486 MB /sec on the four datasets.

Although Extreme Binning runs faster than SiLo-2MB under Inc-set where many duplicate files exist, it runs much slower in other datasets. Since each bin stores all similar files and it tends to grow in size with the dataset size. As a result, Extreme Binning will slow down as the size of each bin increases since each similar file must read its corresponding bin in its entirety. In addition, the design of bin fails to exploit the backup-stream locality that can help reduce disk accesses and increase the RAM utilization by preserving the locality layout in the memory.

Since SiLo uses significantly less RAM space than other approaches for a given dataset, SiLo can also boost the deduplication throughput by caching more index information in RAM to reduce accesses to on-disk index. In fact, the SiLo system can be dynamically configured with users' requirements such as the throughput and duplicate elimination by tuning the appropriate system parameters (e.g., the number of blocks in the cache, the segment size and block size, etc.). Therefore, compared with Extreme Binning and ChunkStash, SiLo is shown to provide robust and consistently good deduplication performance, achieving higher throughput and near-complete duplicate elimination at a much lower RAM overhead.

5 Related work

Data deduplication is an essential and critical component of backup/archiving storage systems. It not only reduces storage space requirements, but also improves the throughput of the backup/archiving systems by eliminating the network transmission of redundant data. We

briefly review the work that is most relevant to our SiLo system to put it in the appropriate perspective, as follows. LBFS [19] first proposes the content-based chunking algorithm with the adoption of the Rabin fingerprints [22], and applies it to the network file system to reduce transmission of redundant data. Venti [21] employs deduplication in an archival storage system and significantly reduces the storage space requirement. Policroniades etc. [20] compares the performance of several deduplication approaches, such as file-level, fixed-size chunking and content-based chunking.

In recent years, more attention has been paid to avoiding the fingerprint-lookup disk bottleneck and enabling more efficient and scalable deduplication in mass storage systems. DDFS [26] is the earliest research to propose the idea of exploiting the backup-stream locality to reduce accesses to on-disk index and avoid the disk bottleneck of inline deduplication. Sparse Indexing [15] also exploits the inherent backup-stream locality to solve the index-lookup bottleneck problem. Different from DDFS, Sparse Indexing is an approximate deduplication solution that samples index for fingerprint-lookup and only requires about half of the RAM usage of DDFS. But its duplicate elimination and throughput are heavily dependent on the sampling rate and chunks locality of backup streams.

ChunkStash [8] stores the chunk fingerprints on an SSD instead of an HDD to accelerate the index-lookup. It also preserves the backup-stream locality in the memory to increase the RAM utilization and reduce accesses to on-disk index. Cuckoo hash is used by ChunkStash to organize the fingerprint index in RAM, which is shown to be more efficient than Bloom filters in DDFS. ChunkStash study also shows that the disk-based Chunkstash scheme performs comparable to the flash-based ChunkStash scheme when there is sufficient locality in the data stream.

The aforementioned locality-based approaches would produce unacceptably poor performance of deduplication in the case of the data streams with little or no locality [3]. Several earlier studies [17, 5, 9, 4] propose to exploit similarity characteristics for small-scale deduplication of documents in the field of knowledge discovery and database. SDS [2] exploits the similarity of backup streams in mass deduplication systems. It divides a data stream into large 16MB blocks and constructs signatures to identify possibly similar blocks. A byte-by-byte comparison is conducted to eliminate duplicate data, which is also the first deduplication scheme that uses similarity matching. But the index structure in SDS appears to be proprietary and no details are provided in the reference paper. Extreme Binning [3] exploits the file similarity for deduplication to apply to non-traditional backup workloads with low-locality (e.g., incremental backup). It stores a similarity index of each new file in RAM and groups many similar files into bins that are stored on the disks, thus it eliminates duplicate files in RAM and duplicate chunks inside each bin by similarity detection.

SiLo is in part inspired by the Cumulus system and Bimodal algorithm. Cumulus is designed for file-system backup over the Internet under the assumption of a thin cloud [18]. It proposes the aggregation of many small files to a segment to avoid frequent network transfers of small files in the backup system, and implements a general user-level deduplication. Bimodal [14] aims to reduce the size of index by exploiting data-stream locality. It merges some contiguous and duplicate chunks, produces a chunk size that is 2-4 times larger than that of general algorithms, and finds more potential duplicate data among the boundaries of duplicate chunks.

Most recently, there have also been studies that explore the emerging applications of deduplication, such as the virtual machines [12, 7], the buffer cache [23], I/O deduplication [13] and flash [6, 11], suggesting an increasing popularity and importance of data deduplication.

6 Conclusion and future work

In this paper, we present SiLo, a similarity-locality based deduplication system that exploits both similarity and locality in backup streams to achieve higher throughput and near-complete duplicate elimination at a much lower RAM overhead than existing state-of-the-art approaches. SiLo exploits the similarity of backup streams by grouping small correlated files and segmenting large files to reduce the RAM usage for index-lookup. The backup-stream locality is mined in SiLo by grouping contiguous segments in backup streams to complement the similarity detection and alleviate the disk bottleneck due to frequent accesses to on-disk index. The combined and complementary exploitation of these two backup-stream properties overcomes the shortcomings of existing approaches based on either property alone, achieving a robust and consistently superior deduplication performance.

Results from experiments driven by real-world datasets show that the SiLo similarity algorithm significantly reduces the RAM usage while the SiLo locality algorithm helps eliminate most of the duplicate data that is missed by the similarity detection. And there exists a solution that optimizes the trade-off between duplicate elimination and throughput by appropriately tuning the locality and similarity parameters (i.e., the size of segment and block).

As our future work of SiLo, we plan to build a mathematical model to quantitatively analyze why SiLo works well with the combined and complementary exploitation

of similarity and locality and learn and adapt to the optimal parameter automatically by the real-time deduplication factor and other system status. Due to its low system overheads, we also plan to apply the SiLo system to other deduplication applications such as cloud storage or primary storage environments that desire to deduplicate redundant data with extremely low system overheads.

7 Acknowledgments

This work was supported by the National Basic Research 973 Program of China under Grant No.2011CB302301, the National High Technology Research and Development Program ("863" Program) of China under Grant No.2009AA01A401 and 2009AA01A402, NSFC No.60703046, 61025008, 60933002, 60873028, Changjiang innovative group of Education of China No.IRT0725, Fundamental Research Funds for the central universities, HUST, under grant 2010MS043, and the US NSF under Grants NSF-IIS-0916859, NSF-CCF-0937993 and NSF-CNS-1016609.The authors are also grateful to anonymous reviewers and our shepherd, Andy Tucker, for their feedback and guidance.

References

[1] AGRAWAL, N., BOLOSKY, W., DOUCEUR, J., AND LORCH, J. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS) 3*, 3 (2007), 9.

[2] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, pp. 1–14.

[3] BHAGWAT, D., ESHGHI, K., LONG, D., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on* (2009), IEEE, pp. 1–9.

[4] BHAGWAT, D., ESHGHI, K., AND MEHRA, P. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining* (2007), ACM, pp. 105–112.

[5] BRODER, A. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings* (2002), IEEE, pp. 21–29.

[6] CHEN, F., LUO, T., AND ZHANG, X. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *FAST11: Proceedings of the 9th Conference on File and Storage Technologies* (2011), USENIX Association.

[7] CLEMENTS, A., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference* (2009), USENIX Association, p. 8.

[8] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference* (2010), USENIX Association, p. 16.

[9] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (2005), ACM, pp. 394–400.

[10] FULL-DATASET. http://en.amazingstore.org/xyj/.

[11] GUPTA, A., PISOLKAR, R., URGAONKAR, B., AND SIVASUBRAMANIAM, A. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In *FAST11: Proceedings of the 9th Conference on File and Storage Technologies* (2011), USENIX Association.

[12] JIN, K., AND MILLER, E. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, pp. 1–12.

[13] KOLLER, R., AND RANGASWAMI, R. I/O deduplication: utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS) 6*, 3 (2010), 1–26.

[14] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 8th USENIX conference on File and storage technologies* (2010), USENIX Association, p. 18.

[15] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th conference on File and storage technologies* (2009), USENIX Association, pp. 111–123.

[16] LINUX-DATASET. http://www.cn.kernel.org/pub/linux/kernel/.

[17] MANBER, U., ET AL. Finding similar files in a large file system. In *Proceedings of the USENIX winter 1994 technical conference* (1994), Citeseer, pp. 1–10.

[18] MICHAEL, V., STEFAN, S., AND GEOFFREY, M. Cumulus: Filesystem backup to the cloud. In *Proceedings of 7th USENIX Conference on File and Storage Technologies* (2009).

[19] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles* (2001), ACM, pp. 174–187.

[20] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the annual conference on USENIX Annual Technical Conference* (2004), USENIX Association, p. 6.

[21] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies* (2002), vol. 4.

[22] RABIN, M. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[23] REN, J., AND YANG, Q. A New Buffer Cache Design Exploiting Both Temporal and Content Localities. In *2010 International Conference on Distributed Computing Systems* (2010), IEEE, pp. 273–282.

[24] TAN, Y., JIANG, H., FENG, D., TIAN, L., YAN, Z., AND ZHOU, G. SAM: A Semantic-Aware Multi-Tiered Source Deduplication Framework for Cloud Backup. In *2010 39th International Conference on Parallel Processing* (2010), IEEE, pp. 614–623.

[25] XING, Y., LI, Z., AND DAI, Y. PeerDedupe: Insights into the Peer-Assisted Sampling Deduplication. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on* (2010), IEEE, pp. 1–10.

[26] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), USENIX Association, pp. 1–14.

G^2: A Graph Processing System for Diagnosing Distributed Systems

Zhenyu Guo[†] Dong Zhou[‡] Haoxiang Lin[†] Mao Yang[†]
Fan Long[‡] Chaoqiang Deng[§] Changshu Liu[†] Lidong Zhou[†]
[†]*Microsoft Research Asia* [‡]*Tsinghua University* [§]*Harbin Institute of Technology*

Abstract

G^2 is a graph processing system for diagnosing distributed systems. It works on execution graphs that model runtime events and their correlations in distributed systems. In G^2, a diagnosis process involves a series of queries, expressed in a high-level declarative language that supports both relational and graph-based operators. Each query is compiled into a distributed execution. G^2's execution engine supports both parallel relational data processing and iterative graph traversal.

Execution graphs in G^2 tend to have long paths and are in structure distinctly different from other large-scale graphs, such as social or web graphs. Tailored for execution graphs and graph traversal operations on those graphs, G^2's graph engine distinguishes itself by embracing *batched asynchronous iterations* that allows for better parallelism without barriers, and by enabling partition-level states and aggregation.

We have applied G^2 to diagnosis of distributed systems such as Berkeley DB, SCOPE/Dryad, and G^2 itself to validate its effectiveness. When co-deployed on a 60-machine cluster, G^2's execution engine can handle execution graphs with millions of vertices and edges; for instance, using a query in G2, we traverse, filter, and summarize a 130 million-vertex graph into a 12 thousand-vertex graph within 268 seconds on 60 machines. The use of an asynchronous model and a partition-level interface delivered a 66% reduction in response time when applied to queries in our diagnosis tasks.

1 Introduction

Distributed applications in data centers are increasingly important as they power large-scale web and cloud services. Often, the execution of such an application involves a large number of cooperating processes running on different machines, spanning multiple software modules and layers, tolerating and recovering from various machine failures and network disruptions. Increases in both the scale and complexity of such systems have made it difficult to understand and diagnose their runtime (mis-)behavior.

Typical diagnosis tasks start with observing misbehavior or anomaly, navigating through runtime information such as logs to find relevant information, and processing the information to infer root causes. For example, starting with a log entry with an error message, diagnosis could find all relevant log entries to infer the root cause for the error. As another example, given two similar jobs that noticeably perform differently, diagnosis could extract related runtime information to identify major differences. Also, it might be difficult to spot problems from a large number of low-level runtime events. A useful practice is to aggregate performance information at an appropriate layer, identify which aggregated component in that layer is problematic, and then drill down into the next layer of details in an iterative process.

Effective diagnosis depends heavily on the ability to correlate runtime events and to leverage these correlations. Previous work, especially those on path-based analysis [14, 7, 13, 8, 19, 26, 18, 27], has largely addressed the important problem of generating and correlating runtime information from executions of a distributed system. Often the difficulty for diagnosis is not due to lack of information, but due to the inability to navigate through and process a sea of information to find out what is relevant.

In this paper, we propose G^2, a distributed graph processing system for storing runtime information of distributed systems and for processing queries on such information. Runtime information is organized as a graph, where vertices correspond to events and edges correspond to correlations between events. Diagnosis then involves an iterative process of writing queries against the graph and analyzing the results of those queries. G^2 provides a declarative language that supports relational and graph operators that operate on the graph structure. For example, given an error log entry e, a G^2 query can be issued to find all events (vertices) that vertex e is causally dependent on, where causal dependencies are captured by certain types of edges. This query uses a *slicing* operator that G^2 provides. From a starting vertex v, *forward slicing* finds all vertices that causally and transitively depend on v, while *backward slicing* finds all vertices that v is dependent on.

Graph aggregation and summarization are another effective way of reducing the amount of information to be examined during diagnosis. In an execution graph, each vertex is associated with a context that indicates the aggregation units that the event belongs to. Examples of aggregation units include static ones such as components, classes, and functions, as well as dynamic ones such as machines, processes, and threads. A G^2 query can aggregate information at an appropriate level. For example, to compare executions of two jobs, a query can compute

the forward slices from the starting points of two jobs. To make comparison easier, the query can continue to compute a machine-level aggregation from the two slices. This requires a *hierarchical aggregation* graph operator that transforms an input graph into a smaller one: it condenses each continuous segment of events with the same aggregation unit (e.g., machine) to create a single supernode and applies an aggregation function on those events to compute the an associated aggregated value.

Distributed query execution in G^2 is supported by a distributed storage and execution system that addresses the challenges of storing and processing large execution graphs with millions or even billions of vertices efficiently. In G^2, events and correlations are captured on local machines as they occur during system execution, leading to a natural partitioning of an execution graph.

G^2's execution engine is tailored for execution graphs that exhibit significantly different characteristics from other large graphs, such as social and web graphs. Execution graphs tend to have long paths corresponding to events along a logically related progression of execution, where social and web graphs have relatively small diameters. Graph operations on execution graphs are often in the form of graph traversal, which is again different from iterative graph operations that must proceed in globally synchronized rounds, such as in page-rank computation for example. Consequently, G^2 embraces *batched asynchronous iterations*, where processing on each partition is batched, but does not have to proceed synchronously in lock steps. Both slicing and hierarchical aggregation fall into this model that allows for improved parallelism and efficiency than the bulk synchronous computation model in previous work, such as in Pregel [24]. Barriers are used only at the end of graph traversal or to create global consistent checkpoints for failure recovery. Furthermore, partitions tend to contain long *local* paths before those paths connect to vertices on other partitions due to cross-machine communication. Graph traversal within each partition is therefore significant to the overall graph traversal performance. Instead of a vertex-oriented interface, G^2 exposes a partition-oriented interface that allows partition-level aggregation states to be maintained in an appropriate data structure. This is particularly valuable for hierarchical aggregation, where the choice of partition-level data structure significantly influences performance.

We have built a prototype and applied it to a set of distributed systems, including Berkeley DB [2], SCOPE/Dryad [11, 22], and G^2 itself. Berkeley DB is a replicated distributed key-value database that can be easily linked with applications. SCOPE/Dryad is a production data intensive computation system, which includes a distributed file system, a distributed execution engine (Dryad), and a declarative query language (SCOPE). G^2 is shown to be effective in diagnosis: for instance, using a query in G2, we traverse, filter, and summarize a 130 million-vertex graph into a 12 thousand-vertex graph within 268 seconds on 60 machines. The optimizations we introduce into G^2's execution engine are effective: the use of asynchronous model and partition-level interface delivered up to a factor of 3 performance improvement when applied to graph operators in our diagnosis tasks. We have also studied scalability of G^2 and the checkpointing overhead introduced to enable failure recovery.

The contribution of G^2 is two-fold. First, as a tool, G^2 enables efficient distributed-system diagnosis by allowing users to write declarative queries with both relational and graph operators, and by providing a distributed engine that executes those queries efficiently. Second, as a distributed system, G^2's execution engine targets a different type of graphs with different structural characteristics and with different type of graph operations. It allows a batched asynchronous graph computation model and a partition-level interface, which have contributed significantly to its efficiency.

The rest of the paper is organized as follows. Section 2 introduces the system execution graph data model, and the diagnosis primitives applied to the graph. Section 3 presents the operators, and the language that G^2 supports, as well as several examples expressed in those constructs. The design and optimization of the distributed graph engine is the focus of Section 4, followed by implementation details in Section 5. We evaluate G^2 and share experience in Section 6. Section 7 discusses the related work. Finally, we conclude in Section 8.

2 MODEL

Distributed-system diagnosis in G^2 centers on the data model and the operations defined on the model, which are the topic of this section.

2.1 Text, Paths, and Graphs

Traditionally, system diagnosis treats runtime information (e.g., logs) as *unstructured* text and involves a tedious and ineffective process of going through logs using primitive text-processing tools such as grep. Using grep on a special tag (such as a request id) captures all entries that are explicitly related to that request, but is likely to miss information that has implicit dependencies.

Previous work [14, 7, 13, 8, 20, 18, 27] on correlating runtime information has effectively addressed this shortcoming by capturing common causal relationship in distributed systems. A *path*-like abstraction is often used to track how a request flows through a distributed system. This relatively simple structure is effective for request-centric analysis and modeling, and reflects a good balance between what an abstraction enables, the simplicity

of an abstraction, and the complexity involved in supporting operations on an abstraction.

Yet, the effectiveness of a path-based model is constrained by its simplifying assumptions: by embracing paths based on requests, the model cuts off interactions between requests that occur in distributed systems. For example, Figure 1 shows a piece of code for a replicated file system. The system receives client requests and appends them in a local cache (line 2-4). When there are enough accumulated requests (line 6), the system batches requests, writes them to local disks (line 11), and forwards them to secondaries for data replication (line 12). The *OnPersistRequests* call in Figure 1 is in fact a batch operation of multiple write requests from clients to the distributed file system. In such a case, it is difficult to assign a path id to the events inside the call (e.g., event *h* can only share with the path id from *e* or *f*, but not both). Two paths might also be correlated when they access the same shared variables. In fact, a more general graph is already used to some extent in previous work such as Pip [26].

G^2 instead explores a different point in the design space. Rather than constraining users to a path-based model a priori, G^2 preserves and presents the full structure captured during the execution of a distributed system as a graph. During diagnosis, users can choose to construct paths from such a graph if paths are appropriate for the diagnosis task at hand, or they can choose to process information in a different way that is more appropriate for that particular task. G^2 does not make that decision for users during the modeling phase. This design choice effectively shifts the burden to the underlying distributed engine, as it must enable efficient operations on a more complicated graph structure.

2.2 Execution Graph

G^2's execution graph model embraces two key concepts: *causality* and *aggregation*. This is based on our observation of common system diagnosis practices: users tend to (i) follow cause-effect relations to find relevant information and (ii) to summarize runtime information at an appropriate aggregation level in a hierarchy in order to find trouble spots for further in-depth analysis.

In an execution graph of G^2, each runtime event from a target system is represented as a *vertex*. In Figure 1, events are shown in small rectangles; examples are *printf* log event *b* (line 3), asynchronous request define events *c* and *d*, and request use event *e*. A context is associated with an event, indicating the aggregation units that the event belongs to. Multiple levels of aggregation units can be defined. Examples include static constructs, such as modules, classes, and functions, as well as runtime constructs, such as machines, processes, and threads.

Runtime events are correlated, where directed edges in an execution graph are used to represent such correlations. Different types of edges can be defined for different types of correlations. For example, an *use* edge connects a source event that defines/forwards an object with a destination event that consumes that object. Network messages or cross-thread requests are examples of such objects. $\langle c,e\rangle$ and $\langle d,e\rangle$ in Figure 1 are use edges. A *sync* edge indicates synchronization of two events from two different threads in order to ensure exclusive access to a shared object or ensure ordered inter-thread execution. A *fall-through* edge connects two consecutive events in the same thread (e.g., $\langle b,c\rangle$).

G^2 provides primitives to define and customize graph traversal for diagnosis. Two are built-in: *Slicing* finds all causally related events in a graph and *HierarchicalAggregate* summarizes information at an appropriate aggregation level.

2.3 Filter with *Slicing*

Instead of simply "grepping" runtime information with a special tag, *Slicing* filters information using graph structure: it starts from a *root event* and transitively collects *causally dependent* events. Forward and backward traversal yield a *forward slice* and a *backward slice*, respectively.

Computing precise and complete causal dependencies for slicing is usually too costly if not infeasible, where a reasonable approximation is often sufficient in practice. A naive way is to consider all *use* and *fall-through* edges as *causal edges*. Our practical experience has shown that fall-through edges often do not imply causal relations. For example, in a typical implementation of message processing subsystem, a thread will continuously accept new incoming messages and call corresponding message handlers. Fall-through edges between two message handler invocations do not represent any meaningful causal dependencies. Such false causal dependencies could render slicing ineffective. All events in the corresponding message handler should however be considered causally dependent on the message-send event. G^2 introduces *causal scope* to specify, for each *use* edge, the set of events that are causally dependent on the source event of that edge. A causal scope consists of a continuous region from the destination event of each *use* edge: all events within that region are causally dependent on the source event; all fall-though edges within that region are considered causal edges. In Figure 1, large rectangle boxes define causal scopes. The shaded area outlines the forward slice from event *a*.

2.4 Summarize with *HierarchicalAggregate*

Aggregation is another effective way of managing a large amount of data, especially with a hierarchy. There are natural hierarchies in distributed systems: a program is

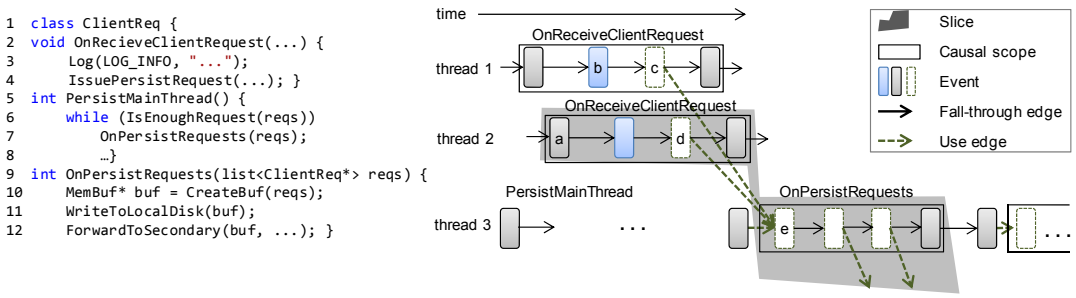


Figure 1: System execution graph, causal scope, and slice.

often made of modules, each module is comprised of classes, and each class contains a set of functions. A distributed-system execution can be aggregated at thread level, then at process level, and further at machine level. A distributed system often consists of multiple logical layers that are application-specific: for example, a system behavior can be analyzed at an RPC layer or at a lower OS layer with a socket interface.

G^2 supports an important notion called *hierarchical aggregation*. The key idea is to construct a condensed graph at an appropriate layer of a hierarchy to summarize system behavior. A continuous segment of events with the same aggregation unit in an execution graph is summarized and condensed into a single higher-level vertex in the resulting graph. G^2 by default attaches signatures of code and runtime location to all events for aggregation. Aggregation in G^2 is customizable: a user can leverage her domain knowledge to specify how to aggregate events and summarize high-level information (e.g., aggregated performance counters) from low-level events.

Figure 2 shows an example of event aggregation when debugging replication in the distributed storage for SCOPE. Numbers inside rectangles are total event-counts within corresponding vertices in the aggregated graphs. An error occurs during a replicated write operation. The upper part of Figure 2 performs event aggregation at machine level: it clearly shows whether the write operation was propagated to all replicas. Once a suspected machine is identified, a user selects that machine and zooms in to see how the write request was processed by each component in this machine, shown in the lower part of Figure 2.

3 Programming in G^2

Programming in G^2 consists of two parts. One is to "program" distributed systems so as to to make them diagnosable by G^2. We defer this to Section 5. The other is for "programming" queries to be executed on G^2, which is the focus of this section.

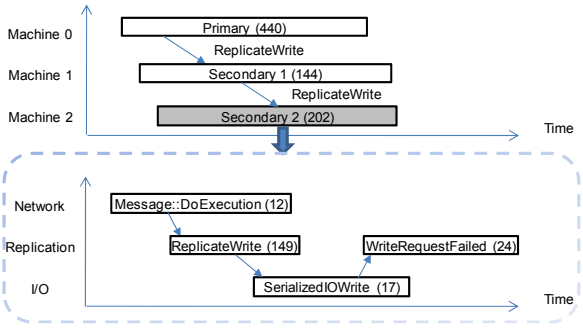


Figure 2: Hierarchical aggregation for a replication implementation. Numbers in rectangles show numbers of events within vertices in the aggregated graphs.

```
1 Graph<TV, TE> Slicing (
2  this Vertex<TV, TE>                    srcVertex,
3  Slice.Type                             type);
4
5 Graph<THighV, THighE>
6 HierarchicalAggregate (
7  this Graph<TLowV, TLowE>                 g,
8  Func<Vertex<TLowV, TLowE>, __out UInt64> labelCb,
9  Func<VertexIterator<TLowV, TLowE>,
10                    __out THighV>   AggreFunc);
```

Figure 3: Graph operators.

3.1 Graph Operators

Figure 3 shows the basic graph operators. Each *Vertex* contains its incoming and outgoing edge lists, and it is a generic type that can be instantiated with $\langle TV, TE \rangle$. Type *TV* describes the data associated with the vertex, such as logs, code locations, and runtime locations, while type *TE* describes the data associated with each edge, such as timestamps for the source and the destination events. A generic *Graph* type can be further defined as a collection of vertices.

As shown in the figure, operator *Slicing* takes *srcVertex* as the root event and *type* as the direction (forward or backward) for slicing. Operator *HierarchicalAggregate* condenses a graph to a higher-level graph (line 5). Given a vertex in the original graph,

```
Events
.Where(e => e.Val.Type == EventType.LOG_ERROR
    && e.Val.PayLoad.Contains("Write request failed"))
.Slicing(Slice.Backward)
.Select(e => Console.WriteLine(e.Val.PayLoad));
```

(a) Error log analysis.

```
var req = Events
.Where(e => e.Val.Location.Name=="SubmitWriteReq");
req.Slicing(Slice.Forward)
.HierarchicalAggregate(
   e => e.Val.Process.Machine.Signature,
   evts => evts.First().Val.Process.Machine.Name)
.CriticalPath(req,dst,e=>{e.Val.SrcTs, e.Val.DstTs});
```

(b) Machine level critical path analysis.

```
var s1 = Events.Where(t => t.VertexID == 1)
.Slicing(Slice.Forward)
.HierarchicalAggregate(...aggregate by component...);
var s2 = Events.Where(t => t.VertexID == 2)
.Slicing(Slice.Forward)
.HierarchicalAggregate(...aggregate by component...);
s1.Diff(s2, e => {e.Val.SrcTs, e.Val.DstTs});
```

(c) Component level performance regression analysis.

Figure 4: Sample diagnosis queries.

the *labelCb* callback returns its aggregation-unit label. The *AggreFunc* callback aggregates a continuous sequence of vertices with the same label (line 9) into a new vertex at the high-level graph (line 10) (Note the structure is determined by G^2, and the associated value(*THighV*) is defined by the callback).

All those operators are built on top of two distributed primitives: *GraphTraversal* and *MapReduce*. *GraphTraversal* starts with a set of vertices in a graph and traverses the graph by following edges forward, backward, or bi-directionally. A user can customize graph traversal by deriving a graph traversal class, which defines computations on vertices, messages passed along edges, as well as final output during graph traversal. *MapReduce* is standard with a map function and a reduce function for aggregation. Details of these distributed primitives and how they are used to build graph operators are left to Section 4.

3.2 Composing Graph Operators

Extensibility and composability are two key features of G^2 design for programming. *Slicing* and *HierarchicalAggregate* both consume a graph and produce another, so they can be composed. We further leverage the extensibility of the LINQ framework [3] in .Net, so that developers can write diagnosis queries using our new operators, LINQ's relational operators, and even customized local analysis modules, such as finding critical path (*CriticalPath*) and comparing two aggregated graphs (*Diff*). Figure 4 shows a set of examples; all from real diagnosis practice.

The first query returns the logs in a backward slice

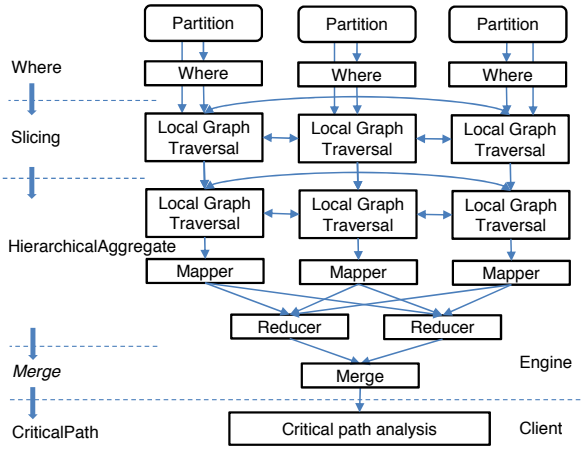


Figure 5: Data flow for the machine-level critical-path analysis query in Figure 4 (b).

rooted from an error log event. The query first uses *Where* in LINQ to locate the error event and then invokes *Slicing*. The second query aims to find straggler machines during processing of a request. The query first calculates the forward slice from the point of request submission, aggregates the slice into a machine-level graph via *HierarchicalAggregate*, and computes the critical path for request processing. Each vertex in the returned critical path summarizes a continuous execution on a machine with the start and stop times of the execution, from which stragglers can be easily identified. The last query intends to find components responsible for an instance of slower-than-normal request processing. It extracts forward slices rooted from the slow request and normal ones, aggregates at the component level, and outputs differences. If needed, users could drill down into problematic components and investigate further at the function level or lower.

4 Distributed Engine

A distributed engine is responsible for transforming diagnosis queries into distributed jobs to be executed on the set of machines storing the execution graphs.

4.1 Overview

In G^2, events and correlations between events are captured and recorded locally, and transformed into appropriate graph representations. G^2 therefore naturally partitions original system execution graphs based on where events occur. Such a partitioning method tends to exhibit good locality as distributed systems are usually designed to minimize cross-machine traffic.

A *job manager* initiates a *job* when a query is submitted. Each machine storing graph partitions runs a *daemon*. The job manager coordinates executions of

phases by communicating with these daemons. A job involves multiple phases that can be represented as a data flow graph. Figure 5 shows the data flow graph for the machine-level critical-path analysis query in Figure 4 (b). It consists of 5 phases: Where, Slicing, HierarchicalAggregate, Merge, and CriticalPath. The distributed engine takes care of the first 4 phases and sends the aggregated results to clients for local critical-path analysis. The Merge phase does not appear in the original query and is added automatically during compilation. Both Slicing and HierarchicalAggregate involve graph traversal, where the latter consumes the graph created by the former and outputs an aggregated graph for client analysis. In particular, the mappers during HierarchicalAggregate shuffle the vertices according to which high level vertex they belong to, and the reducers aggregate the vertices inside one high level vertex using the *AggreFunc* callback provided by the queries.

The part of the data flow graph without graph traversal is similar to directed acyclic graphs (DAG) in previous data-parallel computation engines, such as Map/Reduce and Dryad. Graph traversal however requires a different type of coordination to support loops and barriers. G^2's graph traversal support distinguishes itself from previous graph engines (e.g., Pregel [24]) in several noticeable ways. First, for operations such as slicing and hierarchical aggregation, G^2 supports batched asynchronous iterations, where partitions batch operations locally, but do not have to be synchronized using a barrier in each iteration. Second, G^2 exposes a partition-level interface, rather than a vertex-level interface, to allow better batching and aggregation for graph computation. This is particularly important for enabling efficient implementation of hierarchical aggregation. These optimizations can be applied not only to G^2 but also to other distributed graph traversal problems such as shortest path computation.

4.2 Batched Asynchronous Iterations

A typical graph engine implements *synchronous iterations* through loops and barriers. For graph computation such as page-rank computation and belief propagation, all participants must synchronize with each other in each iteration via a barrier, and in each iteration the participants can only traverse one hop. Such synchronization is easily done with the help of a job manager.

In G^2, we observe that graph traversal for slicing and hierarchical aggregation is inherently *asynchronous*. Take forward slicing for example, each partition has a set of vertices to start with in each iteration (except the first one where only one partition has the root vertex). For one local iteration, a partition starts graph exploration from those vertices following causal edges until it reaches cross-partition edges without synchronization

```
IQueryable<T> GraphTraversal<TWorker> (
 this Graph<TV, TE>         g,
 IQueryable<Vertex<TV, TE>> startVertices
) where TWorker : GPartitionWorker<TV, TE, _, T>;
class GPartitionWorker<TV, TE, TMsg, T> {
 Vertex<TV, TE> GetLocalVertex(ID VertexID);
 void SendMessage(ID VertexID, TMsg msg);
 void WriteOutput(T val);
 virtual void Initialize(VertexIterator<TV, TE>)=0;
 virtual void OnMessage(Vertex<TV, TE>, TMsg) = 0;
 virtual void Finalize() = 0;
};
```

(a) *GraphTraversal* interface.

```
class GPartitionSlicingWorker<TV, TE>
: GPartitionWorker<TV, TE, bool, Vertex<TV, TE>> {
 HashSet<ID> VisitedVertices;
 void Initialize(VertexIterator<TV, TE> inits) {
   foreach (var v in inits)
     SendMessage(v.ID, true);
 }
 void OnMessage(Vertex<TV, TE> v, bool msg) {
   if (VisitedVertices.Contains(v.ID)) return;
   VisitedVertices.Add(v.ID);
   WriteOutput(v);
   foreach(var e in v.OutEdgeIterator)
     if (e.IsCausal())
       SendMessage(e.DstVertexID, true);
 }
 void Finalize() {}
}
```

(b) *GPartitionSlicingWorker* for forward slicing.

Figure 6: *GraphTraversal* interface and example

with others after every one hop traversal. When this iteration ends, a partition reports to the job manager with pointers to lists of vertices to other partitions for further exploration. The job manager will notify other partitions of the availability of these lists. A partition finishing the current iteration can fetch the lists of new vertices from other partitions and start the next iteration. It does not have to wait to get lists from all other partitions before initiating the next iteration.

G^2 does support global barriers for two cases. In the first case, completion of a graph-traversal stage is through a global barrier: all participants must have completed their last iteration locally. The job manager initiates the next phase of computation only after that global barrier is established. In the second case, the job manager can periodically introduce a barrier to an ongoing graph-traversal stage for failure recovery; the barrier is used to perform a globally consistent snapshot.

4.3 Partition vs. Vertex

G^2 provides a GraphTraversal interface so that users can implement their own custom graph-traversal algorithms. Previous graph processing systems such as Pregel allow users to specify actions on each vertex, which is natural for a large number of graph computation algorithms. However, we have found a partition-level interface offers additional opportunities for better performance.

Graph Traversal Interface. Figure 6 (a) shows the signature of *GraphTraversal*. It starts from a set of initial vertices (line 3), with traversal polices designated by *TWorker* derived from *GPartitionWorker* (line 4). When a graph traversal phase starts, G^2 creates an instance of *GPartitionWorker* on every graph partition of *g* (line 2), and the job manager coordinates the workers to perform multiple iterations of computation: a first round for *Initialize* (line 9), followed by multiple rounds of graph traversal via message exchanges among vertices (line 10) until all workers reach the completion barrier, and a last round for *Finalize* (line 11). In each round, a worker creates remote messages for other partitions. Those remote messages are eventually transported to appropriate partitions and serve as the input for next-round computation on those partitions.

Forward Slicing. Figure 6 (b) shows a sample that implements forward slicing. During *Initialize*, the worker sends a message to the initial vertices of the graph traversal via *SendMessage* (lines 5,6). After initialization, each worker invokes *OnMessage* (line 8) on each message, inside which a worker can read/write partition-local states (lines 9,10), produces partial outputs via *WriteOutput* (line 11), and send messages to other vertices via *SendMessage* (line 14) by following the edges of the current vertex (line 12). *OnMessage* does *not* cause a real network message to be sent: for a local destination, the worker again applies *OnMessage* on the destination vertex in the current round. Only messages destined to a remote partition are gathered and made available to other partitions at the end of this iteration. After a worker completes the current round, it fetches the available remote messages for its partition from other partitions, and starts a new round. This process ends when all workers have completed the current rounds with no new remote messages. In the final round, the workers invoke *Finalize* (line 16), which usually produces outputs of this traversal phase from final local states. It is empty in this case because output is generated during traversal (line 11).

Hierarchical Aggregation. The value of exposing a partition-oriented interface is more evident in the implementation of *HierarchicalAggregate*.

Figure 7 (a) shows a simple vertex-oriented implementation from a vertex's perspective. Each vertex has a label based on its context; for example, the label is its process id for process-level aggregation. We use *AggId* to identify a set of vertices that have already been aggregated together: those vertices will have the same value *v.AggId*. Every vertex uses its own *ID* as the initial *AggId* (line 3), and broadcasts both its label and *AggId* to its neighbors (lines 4,5). A message is ignored when a receiving vertex has a different label (line 8), indicating a boundary of aggregation. Otherwise, if an incoming *AggId* is smaller than the current one, a vertex changes its own *AggId* and propagates the change to its neighbors (lines 9-12). The traversal ends when all vertices are assigned the smallest label of the vertices to be aggregated together.

```
1  void Initialize(VertexIterator inits) {
2    foreach (Vertex v in inits) {
3      v.AggId = v.ID;
4      foreach (Vertex iv in neighbour vertices)
5        SendMessage(iv, {v.ID, v.Label});
6    } ...
7  void OnMessage(Vertex v, MSG msg) {
8    if (msg.Label != v.Label) return;
9    if (msg.AggId < v.AggId) {
10     v.AggId = msg.AggId;
11     foreach (var e in connected edges)
12       SendMessage(e.DstVertexID, msg);
13   } ...
```

(a) Vertex oriented implementation.

```
1  Map<ID, ID>   VertexLeader;  // vertex->leader
2  Map<ID, ID>   LeaderAggIds;  // leader -> aggId
3  Map<ID, ID[]> RemoteVertexGroup;//leader->rvertices
4  void Initialize(VertexIterator inits) {
5    ... local aggregation to initialize the maps ...
6    ... send messages to remote vertices ...
7  }
8  void OnMessage(Vertex v, MSG msg) {
9    if (msg.Label != v.Label) return;
10   ID leaderId = VertexLeader[v.ID];
11   int oldAggId = LeaderAggIds[leaderId];
12   if (msg.AggId < oldAggId) {
13     ... update aggId for the group ...
14     foreach (var vid in RemoteVertexGroup[leaderId])
15       SendMessage(vid, msg);
16   } ...
```

(b) Partition oriented implementation.

Figure 7: Optimization for *HierarchicalAggregate*.

Figure 7 (b) shows a partition-oriented implementation, where partition-level aggregated states are maintained (lines 1-3) and initialized (lines 5-6). These data structures essentially aggregate local continuous segments with the same labels and update them as a single unit during traversal, rather than going through each vertex repeatedly: each local continuous segment with the same label is assigned a leader. Rather than having each vertex maintaining an *aggId*, the partition maintains a mapping from leaders to *aggId*s in *LeaderAggIds*. Because vertices with the same leader always have the same *aggId*, a partition can simply update one entry in *LeaderAggIds* for all those vertices when the *aggId* changes for any of the vertices. Similarly, destination vertices of cross-partition edges from this segment of vertices are recorded in *RemoteVertexGroup* and can be identified without following the edges within this segment repeatedly. Those data structures are populated during initialization. When a message arrives at a partition with a boundary vertex as the destination vertex, the worker checks its label (line 9), and updates the *AggId* of the corresponding leader vertex if it receives a smaller *AggId* (lines 10-13). Finally, it broadcasts the new *AggId*

Component Name	Language	LOC(K)
Annotation Library	C++, C	3.4
Binary Rewriter	C++/CLI	1.6
Transformer	C++	1.5
Engine(JobMgr,Daemon,MetaSvr)	C++	27.5
FrontEnd(Compiler, JobClient)	C#	17.3
VS AddIn(Wizards, UI)	C#	57.8
Total	-	109.1

Table 1: Components in G^2.

to its cross-partition neighbors (lines 14-15).

4.4 Failure Handling

G^2 supports iterative graph computation, which renders inapplicable the map/reduce type of failure recovery using re-computation. In a synchronous graph computation model, where a global barrier is established at each round, a globally consistent checkpoint can be taken at each barrier. When failures happen, computation can be rolled back to the most recent checkpoint. G^2 allows each partition to maintain states. A checkpoint therefore covers such per-partition state, as well as the remote messages that each partition generates at the end of a round for other partitions. Appropriate levels of redundancies might be needed for checkpoints in order to recover from permanent machine failures.

With an asynchronous graph computation model, G^2 can choose to insert a global barrier at an appropriate interval for consistent checkpointing. We can also resort to the standard Chandy-Lamport algorithm for taking a consistent snapshot, where a global barrier is a special simple (yet less efficient) implementation for this algorithm. In the worst case, G^2 can always roll back to re-execute a graph-traversal phase (assuming that failures do not lead to data loss in the original graph information). Our current implementation uses global barriers for consistent checkpointing, but do not replicate the checkpoint to tolerate permanent failures.

5 Implementation

G^2 provides a complete tool set to help developers diagnose systems. Table 1 shows the programming language and lines of code for components of G^2: annotation library and binary rewriter are used to capture system execution graphs. Transformer is used to store a graph. Engine, Front-End, and Visual Studio AddIn are for processing and visualizing a graph.

Capture Graph. G^2 can use existing traces from previous work, e.g., those on path-based analysis, to build execution graphs. It also provides its own tool chain for developers to instrument target systems for gathering information of interest. Whether instrumentation requires manual code change depends on the types of edges to be captured. We have developed a Phoenix [5] based binary rewriter tool to annotate *synchronous use* edges (i.e., call) and their corresponding causal scopes (i.e., the call boundary) automatically. A user can choose what to instrument with a configuration file, reflecting her choice to balance between cost and coverage. G^2 also captures *sync* edges automatically at the Win32 layer by instrumenting Windows synchronization APIs. For *asynchronous use* edges, G^2 provides an annotation library; the following code illustrates how to track network messages and their corresponding handlers using this library. Users first annotate a context(*NetworkMsg*) by making it inherit a *G2::CausalCtx* object. Users then add a call to *LogUseEdgeBegin* and *LogCausalScope* respectively, when this context is about to be delivered and used. The macro call of *LogCausalScope* is a C++ object, whose lifetime defines a causal scope.

```
1 class NetworkMsg : public G2::CausalCtx {
2   int Send(...) {
3     LogUseEdgeBegin(this, ...);
4     ...
5   } ...
6 void OnRecvNetworkMessage (NetworkMsg& msg, ...) {
7   LogCausalScope(msg);
8   msg.Execute(...);
9   ... }
```

Store Graph. We developed a *Transformer* that converts raw runtime-event streams to database tables. G^2 stores a system execution graph in four relational tables. Each object has a unique key, which we use as reference keys across tables and partitions. The *CodeLocation* provides the context for each event and in particular the component, class, function, file, and line number of the statement that generates the event. The *ProcessInfo* table covers the runtime process information, such as process id, machine name, process start time, and so on. The *Event* table contains information about each event, including its type, a reference to an edge if it is an endpoint of that edge, a physical timestamp, references to *CodeLocation* and *ProcessInfo*, and payload (e.g., *printf* log content). The *Edge* table contains the edge type, a unique edge ID, and references to the source and destination events. The *Edge* table defines the *structure plane* of a system execution graph, while the other three form the *data plane*. A slicing operation can be done purely on the *Edge* table, but for event aggregation it is often necessary to query the *CodeLocation* and *ProcessInfo* tables. The *Edge* table is frequently accessed during graph traversal and is therefore cached in memory for fast access.

Process Graph. Queries submitted to G^2 are compiled into a distributed query plan, with appropriate resource files dispatched to workers running on machines managing partitions of a graph. Execution of a query plan is done through coordination between the job manager and the workers, as described in Section 4. The job manager

Systems	Acs#	Ace#	Func#	Rule#
G^2	9	11	197	10
SCOPE/Dryad	17	13	730	5
BerkeleyDB	2	2	1,542	23

Table 2: Instrumentation statistics. *Acs#*, *Ace#*, *Func#*, and *Rule#* refer to the number of manually annotated causal scopes, manually annotated edges, instrumented functions, and rules in the configuration files for the binary rewriter, respectively.

monitors progress of graph traversal and assists in message exchanges between workers. After a round of local processing ends on a partition *A*, the worker for *A* groups messages based on their destinations and notifies the job manager of the list of partitions with data from *A*. The job manager piggybacks the list partitions with data ready for *A*. The worker for *A* will then fetch those from the corresponding workers. To make message exchange efficient, workers cache generated message groups in memory and discard them after they are fetched. The job manager is also responsible for enforcing global barriers upon completion of graph traversal, as well as to create consistent checkpoints.

6 Experiments and Experience

We have applied G^2 to SCOPE/Dryad, G^2 itself, and BerkeleyDB. Our evaluation attempts to answer the following questions: a) what is the cost of applying G^2? b) how does the G^2 engine perform on real execution graphs? c) does G^2 help developers diagnose complicated distributed system problems?

Target systems are co-deployed with G^2 on a cluster of 60 machines; each has a dual 2GHz Intel Xeon CPU, 8 GB memory, two 1TB SATA disks, and are connected with 1 Gb Ethernet.

6.1 Cost of Applying G^2

Human effort. Table 2 reports the statistics about the annotation effort to apply G^2 on these systems. For instrumenting functions, users write only a configuration file for the binary rewriter to specify names of functions they are interested in. For all three cases, the configuration files are less than 25 lines, as shown in Table 2.

The *asynchronous use* edges and correspondent causal scopes require manual annotation on source code. Our experiences show that most asynchronous messages and events are handled by a small number of components or by a middleware library, making annotation easy. We annotated fewer than 20 places for each benchmark. Annotations on G^2 took us less than one hour. Interestingly, in our SCOPE/Dryad experiment, we did forget to annotate a place where the code directly uses the *CreateProcess* function to create a new process, bypassing the middleware component that we annotated. Such cases are rare and can often be discovered during a diagnosis process. Developers can optionally capture other dependencies: we do not model those.

Runtime overhead. Runtime overhead for emitting events and edges is comparable to those in previous work on capturing causal dependencies [14, 7, 13, 8, 19, 26, 18, 27]. There are several categories of events/edges. The first category are asynchronous use edges and the corresponding causal-scope events (e.g., message send/receive), which are always captured. The second are legacy *printf* logs. These two parts do not introduce noticeable system slowdown compared to previous systems (with the same *printf* logs). The third are events from instrumented functions, and the cost is proportional to the numbers of function invocations that are captured. Usually invocations of interface functions of each component that are associated with error and failure handling are enough for diagnosis. For our experiments on the three distributed systems, only less than 0.1% of overall function invocations are captured in system execution graphs, and no noticeable overhead was observed. If more functions need to be instrumented, and we cannot afford to instrument all, we can turn to dynamic instrumentation techniques as done in our previous work [23].

Table 3 reports the statistics on sample execution graphs from our target systems. The G^2 data include events from executing tens of diagnosis queries against a SCOPE/Dryad snapshot. The SCOPE/Dryad data came from ten SCOPE queries for calculating different statistics of web data. The BerkeleyDB data was collected during approximately one hundred instances of system initialization guided by a model checker; the goal is to use G^2 to assist model checking research in another project. All numbers reported are per-machine averages. For example, for SCOPE/Dryad, a 120-minute trace generates about 1.2GB of G^2 data on each of the 60 machines. On average, the imposed I/O bandwidth ranges from 85.3 KB/s (G^2) to 174 KB/s (SCOPE/Dryad) on average, which did not cause noticeable runtime interference to the host systems. Not reported in this figure, in our sample SCOPE/Dryad execution graph, about 28% of the events recorded are legacy logs (category 2), which account for 64% of total sizes. Category 1 accounts for 33% by count and 16% by size, while category 3 takes the rest.

6.2 Performance Evaluation

This section evaluates the performance of G^2 engine.

Graph statistics. Table 3 shows the numbers of edges and vertices for the sample execution graphs from our target systems. The number of edges is far fewer than the number of events because all fall-through edges are implicit. The func# is the number of invocations for instru-

Systems	LOC(K)	Func#	Edge#	Event#	Raw(MB)	DB(MB)	Time(min)	node#
G^2	27	267,728	634,704	1,212,778	85	231	17	60
SCOPE/Dryad	1,577	3,128,105	8,964,168	20,106,457	1,226	3,269	120	60
BerkeleyDB	172	46,164	92,502	186,597	14	29	2	3

Table 3: Execution graph statistics about a snapshot for the target systems.

mented functions as specified in Table 2. System execution graphs can be large: the SCOPE/Dryad snapshot has on average more than 20 million events on each machine. The database size (DB) is approximately three times the raw event stream size (Raw), due to verbose DB data format (factor of 1.5) and associated indices. The *edge* table (including its indices) counts for only 30% of the total database size. Because it is frequently accessed, caching it in memory makes sense.

End to end performance. We evaluate the end to end performance of G^2 with 5770 random queries on the SCOPE/Dryad graph. Each query calculates a forward slice from a randomly selected root event and then computes a process-level aggregation on the slice. Figure 8 shows the overall running time of these queries with all optimizations turned-on. The G^2 engine is generally fast on these queries: 94.5% queries finished within 5 seconds, with only three queries (in the upper-right circle) taking more than 100 seconds. Our investigation shows that the randomly chosen root events for those three queries are close to the entry point of the Dryad job, yielding huge slices with up to 130 millions of events. Running time depends not only on the sizes of resulting slices, but also on properties of the graph (and its partitions), those properties dictating concurrency of query processing. For example, some queries (in the circle on the left) take more than 50 seconds, even though the corresponding slices are relatively small. This is because they experience a period of time with low concurrency.

We also inspected the result of hierarchical aggregation. The result shows that it is effective in simplifying graphs. All resulting process-level graphs contain less than 85 vertices, except the three queries in the upper right circle: the aggregation yields graphs with only 0.01% of vertices.

Graph and graph computation characteristics. We recorded and examined a large forward-slice computation on the SCOPE/Dryad graph. Figure 9 shows how the numbers of events, local edges, and remote edges vary over time. The SCOPE/Dryad job has a bootstrap phase, during which a job scheduler copies resource files between machines. In the actual execution, this phase takes little time. However, because this phase involves a series of communication, slicing at this segment of the execution graph has little concurrency. It takes a relatively long time to process even though the number of events and the

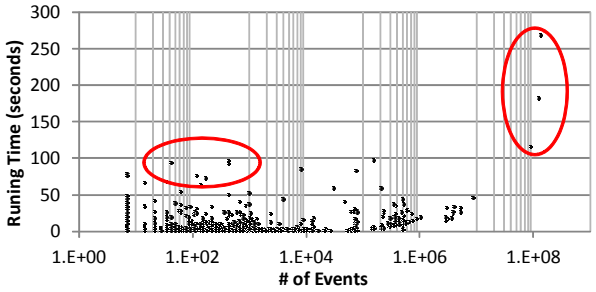


Figure 8: Process level aggregation performance.

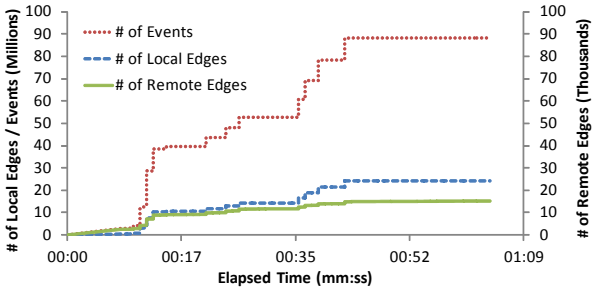


Figure 9: How the # of events, local edges, and remote edges vary along the execution time.

total I/O are small. This is reflected in the flat start in the figure. Graph traversal experiences respectable concurrency in the middle range when traversing the portion of the graph for the real Dryad execution. Then after around time 00:40 it starts to process the portion of the graph corresponding to the final phase, in which a job manager again has to talk to many machines to fetch statistics and to write them into a distributed file. The overall concurrency level on 60 machines is 7.23.

Effectiveness of graph engine optimizations. To evaluate the effectiveness of batched asynchronous iteration and partition-oriented interface design, we measured both *Slicing* and *HierarchicalAggregate* performance with nine different configurations, which are the combinations of two dimensions of configurations. The first is whether to enable *B*arriers among workers and *C*heckpointing after each round (*None*, *B*, or *B*/*C*). The second is to choose which local graph traversal policy (*OneHop*, *Batched*, or *Partition*). *OneHop* is to allow one hop traversal only in each round, a typical setting for other graph engines such as Pregel; *Batched* is to tra-

Phase	OneHop	Batched	Partition
Slicing(None)	1.49	0.99	1.00
Slicing(B)	2.70	1.19	1.21
Slicing(B/C)	2.86	1.27	1.27
Aggregation(None)	1.80	1.48	1.00
Aggregation(B)	2.67	1.91	1.01
Aggregation(B/C)	3.40	2.42	1.09

Table 4: Relative execution time for the component level aggregation analysis with nine different configurations. Abbreviations: B - barriers are enabled among workers; B/C - barriers are enabled among workers, and partition state is checkpointed.

verse until no further local vertices to be visited during this round; and *Partition* is similar to the second, but with the partition-state optimization discussed in Section 4.3, enabled by G^2's partition-oriented interface. Table 4 shows the average relative execution time for the component level aggregation analysis for a Dryad job with the nine configurations; each runs ten times. To be fair, with $\langle OneHop, B/C \rangle$, we checkpoint every 7 and 18 rounds for slicing and aggregation, respectively, so that they take approximately the same number of checkpoints as in other configurations.

Overall, batched asynchronous iterations and partition-oriented interface are effective: without checkpointing, we see a 62-63% reduction in latency for both slicing (2.70 vs. 1) and hierarchical aggregation (2.67 vs. 1). The data also reveal the following: (i) Batched asynchronous iterations bring benefits in two ways: First, it allows local traversal to proceed (as in *Batched*) and significantly reduces the number of global rounds (from 208 rounds to 28 rounds for Slicing and from 111 rounds to 6 rounds for hierarchical aggregation). Second, it removes the need for global barriers. This is particularly effective when there are many rounds and significant variations across machines in each round. It is noticeably ineffective for Aggregation ($\langle Partition, B \rangle$, $\langle Partition, None \rangle$) because our partition-oriented optimization makes process-time variations between partitions negligible. (ii) Partition-oriented interface and data structures are effective for Aggregation (with 32% reduction) because we are seeing large local islands (e.g., one island with 7.7 million internal edges and only 2,895 remote edges): those local islands do not have to be visited repeatedly with our optimization. (iii) Overhead of checkpointing depends on how frequent we checkpoint and how much data we checkpoint. OneHop introduces lower overhead (5.11 MB/s) because it checkpoints the same amount of state in a longer time period compared to Batched (7.18 MB/s), and Batched has higher overhead compared to Partition because its state size is larger than that under Partition (5.96 MB/s).

Scaling performance. We evaluated scaling performance from two perspectives. The first is to measure scaling in terms of the number of machines. We do not show figures due to space constraints. We observe that the job latency decreases almost linearly initially when more machines are used. But after we have more than 16 machines, the speedup slows down due to inherent limit on concurrency. With 60 machines, the average latency is reduced to under 3 minutes from over 14 minutes on 8 machines.

The second is to measure scaling in terms of the number of concurrent queries. We use two different sets of slices for those queries. The first set has several large slices, which involves 5 to 8 graph partitions and contains approximately 0.4 to 1 million events. The second set has a set of randomly selected slices, which typically involves 1 to 2 machines and contains several thousands of events. For large slices, latency increases dramatically when we reach 100 queries. For small slices, the system can support almost 500 queries simultaneously without affecting query latencies and can handle 5,000 queries with an average latency under 3 minutes.

6.3 Experience, Limitations and Future Work

To make G^2 accessible to developers, we have built a set of templates to guide the use of the system and integrated the tool into Visual Studio for a seamless debugging experience. The *Visual Studio AddIn* includes a set of common diagnosis tools based on G^2, including event navigation along edges in all levels of graphs (called *gwalker*), as well as a set of wizards focusing on specific visualized diagnosis tasks, such as error log analysis, critical path analysis, and performance regression analysis, as discussed in Section 3.2. Following are two showcases from our experiences.

Slower job. When developing G^2, we found that query processing time became 60 times slower after minor code changes. To investigate, we applied G^2 for a machine-level aggregation with critical-path analysis on a forward slice of a query job. The result showed that most of the processing time was spent on machine *srgsi*-10. We then zoomed into a thread-level execution graph, and performed a graph diff with its counterpart from the same run in the previous day (before code changes were applied). Figure 10 depicts the diff result, which shows that thread 8772 has the largest deviation between the two jobs in terms of execution time. We further zoomed into the component level and then function level execution graphs in that thread, and found that the largest deviation happened in the *AcquireRowById* function which reads a row from a local table using the primary key. Our further investigation revealed that the table did not have a proper index, the result of a bug we introduced in Transformer that caused index creation to fail. This kind of perfor-

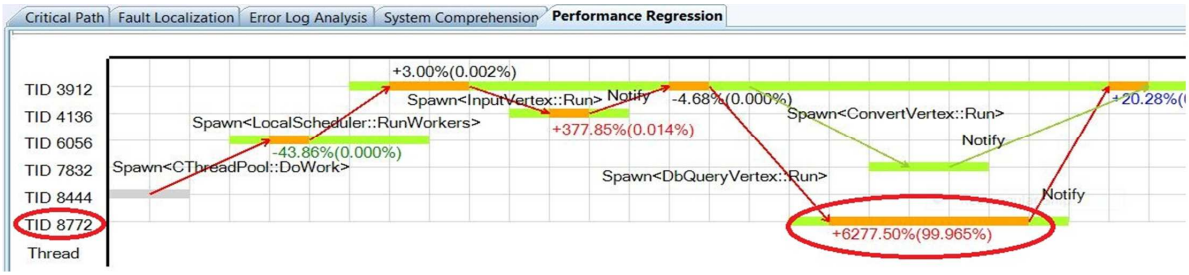


Figure 10: Thread level performance regression diff.

mance regression problem is common in our experiences and a hierarchical diff analysis between two similar tasks is often effective in identifying root causes.

Failed write. We have used G^2 to investigate root causes of error logs in SCOPE/Dryad. One case is shown in Figure 11. The error log in the underlying distributed storage system reported that a write request to a chunk server *c1* at time 21:27 failed. Using G^2, we managed to find its root cause through several steps, marked in Figure 11.

1. We computed a backward slice starting from the error log entry and listed the warning and error entries ordered by time (*OrderBy*). A warning log entry showed up in the disk IO module, indicating that a chunk *x* was marked as deleted. However, we were not able to figure out why this happened based on the information in this backward slice.

2. We wrote a *Where* query for the most recent logs on the same machine who contain keyword "chunk *x*", trying to connect the missing edges which may tell why chunk *x* was marked as deleted. The query returned an event *B* indicating that at time 21:17 a background thread marked this chunk as deleted.

3. We use *gwalker* to navigate the logs on the graph from event *B*, and found an event *C* indicating that the meta server *m1* sent a delete request to *c1* because it found that chunk *x* no longer belonged to any file stream.

4. To locate the origin of the write request to *c1* on presumably deleted chunk *x*, we aggregated the backward slice rooted at event *A* at process level and found the write request came from *c3*, and propagated by *c2*, where c1, c2, and c3 formed a replication group for *x*. A further drill-down of the logs on *c3* showed that *c3* restarted at 21:27. During its replication log replay, it found an incomplete local chunk *x* and issued an empty write request to sync data from other replicas (*c1* and *c2*).

5. Trying to understand why *c3* did not receive the delete request from meta server for chunk *x*, we ran a process level aggregation on the forward slice from event *C*, and found that *m1* sent a delete request to *c3* at time 21:25, but *c3* was not online at that time. This revealed the root cause.

This interactive diagnosis process involved gwalker, slicing (at a specific layer), aggregation, and relational queries in G^2, and is guided with human expert knowledge. It is worth pointing out that G^2, as any diagnosis tool, is not intended to replace human completely. Rather, its value lies in its ability to allow users to find the right information efficiently.

Implicit dependencies. The backward slice in step 1 of the Failed Write diagnosis did not contain all the interesting events for us to find the root cause, due to implicit dependencies (through chunk *x*), which are not captured by G^2. This is a common limitation in causality-based approaches. We managed to connect the dots through a relational query in this case. In our performance diagnosis experience, we also found a lot of problems caused by resource contention or interference, and again such causal relations are not modeled by G^2. In the future, we plan to incorporate some interference analysis techniques (e.g., [25]) to introduce *interference edges* into our model.

Customized slicing computation. We found some slices were fairly large and took a long time to compute. In many cases, users do not need all the information in a slice. To better control the cost, G^2 provides three additional parameters to the *Slicing* operator: maximum slice radius (hops starting from the root event), maximum network hops, and a customizable edge filter which decides whether the computation should continue following this edge for a bigger slice. Our experience shows these parameters can greatly improve the productivity, especially when people are familiar with their target systems.

Deployment and interference. The data placement has three reasonable choices as we see: data on each machine, on one dedicated machine per pod (in which the machines share a same uplink), and on a single machine. We did not run the second option (in a real data center) yet, and our scalability study above touched on diagnosis performance vs. number of machines. Our belief is that the second option is more suitable for a real deployment to minimize interference, while the other two can be used in testing environments depending on the size of the setup.

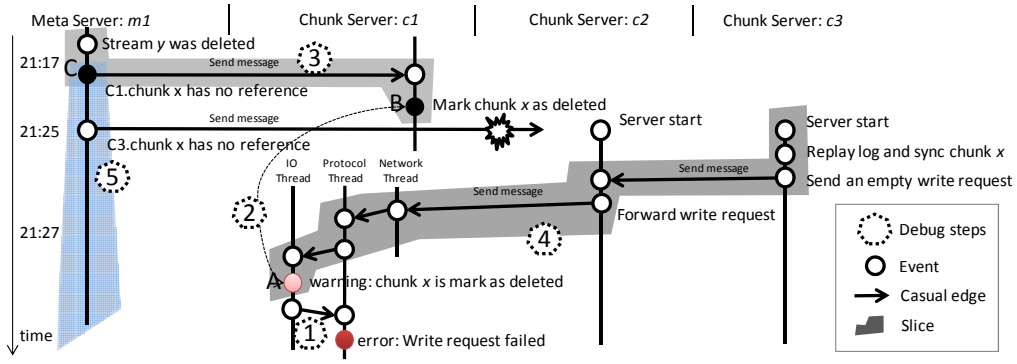


Figure 11: Diagnosis process for the failed-write scenario.

7 Related Work

The design of G^2 draw inspirations from a great number of previous works. We discuss those in three categories: execution modeling, distributed execution engine/storage, and diagnosis platform.

Execution Modeling. G^2 captures overall system behavior in a system execution graph with the help of annotation and instrumentation. It is clearly related to path-based analysis [14, 13, 26, 7, 8, 27], where a path is often defined as a sequence of events that is triggered by a client request. Path instances can either be collected through annotation [14, 13, 26] and schemas [8] provided by developers, or statistically inferred from inter-machine communications [7]. A variety of analysis can be enabled on path instances. For example, Magpie [8] aims to analyze workload models from path instances; PinPoint [14, 13] uses statistical methods to find components that are highly correlated to failed requests; Pip [26] checks these instances against specifications of expected system behavior defined by users. Path instances correspond to forward slices from the points where client requests are submitted in a system execution graph—G^2's model is general in that slicing can be in both directions from any point. Technically, path-based techniques [14, 13, 26, 7, 8, 19] could be applied on these forward slices and therefore integrated into G^2. We plan to investigate this feasibility in the future. X-Trace [19, 18] is similar to G^2 as it captures system behavior as task trees, and it tends to store the information in a service like OpenDHT to allow further distributed processing.

A large body of work focuses on diagnosing distributed systems using purely legacy logs. For example, Wei et al. [29] use machine learning to mine console logs to detect large-scale system problems, and SherLog [31] uses a constraint solver with information from a log to rebuild system execution flow that produces the same log. While no annotation or schema input from users are needed in those systems, there is usually a trade off: a machine-learning approach [29] involves sufficient art to ensure accuracy, while recovering information using constraint solving [31] faces scalability challenges.

Our slicing concept is inspired by program slicing [6] in program analysis. Hierarchical aggregation is related to hierarchical dynamic slicing [28], although the underlying techniques are different. Program slicing captures fine-grain data/control dependencies among variables/statements, and semantic hierarchy inside programs, while G^2 captures dependencies at a coarse granularity and resorts to approximation for scalability.

Distributed Execution Engine and Storage. G^2 hinges on its graph traversal engine to operate on huge execution graphs, often composed of millions even billions of vertices. Pregel [24] is a system for general large-scale graph processing. Tailored for execution graphs and graph traversal, G^2 adopts a batched asynchronous model rather than a bulk synchronous model in Pregel; it exposes a partition-oriented interface, rather than a vertex-oriented one in Pregel.

Other distributed computing engines have also been applied to specific computation on large graphs. Distributed execution engines such as MapReduce [16] and Dryad/DryadLINQ [22, 30] have been applied to compute PageRank [4] on a web graph. Recently, MapReduce Online [15] is also used for interactive big data analysis. G^2 also leverages *MapReduce* as the basic construction primitive to implement the diagnosis operators. Besides, it employs dedicated graph traversal primitive to reduce the query latency. G^2 partitions a graph into partitions and stores partitions on different machines. Distributed storage systems, such as key/value stores or table-based stores, have been studies extensively, although not for storing large graphs in particular. Recent examples include Cassandra [1], Dynamo [17], and BigTable [12]. G^2 adopts the similar approach, and it co-locates the execution to the partitions so as to reduce the storage access latency.

Diagnosis Platform. Several previous diagnosis tools have also leveraged the power of distributed systems.

Cloud9 [10] has pioneered the concept of *testing as a service*. In particular, it shows a symbolic execution testing engine that can be parallelized in a cloud. We share the same vision and believe G^2 can enable diagnosis as a service. Wei et al. [29] parallelized their algorithm for learning legacy logs on Amazon EC2 with Hadoop [21].

Dapper [27] is a tracing framework designed for low overhead, application transparency, and ubiquitous deployment. Trace data are organized in a Dapper trace tree, where each node represents a basic unit of work called a span. Each trace is stored in BigTable. It also offers a programmatic API, as well as an annotation API. Due to its more restricted trace-tree model, it does not support graph-traversal or any of the operators in G^2. DTrace [9] is another tracing framework that supports on-demand instrumentation of distributed systems. It allows customized predicates and aggregation functions via a scripting language. The aggregation is applied to a set of flat trace records, which is different from G^2 as the later applies aggregation to a graph.

8 Concluding Remarks

Execution graphs capture runtime behavior of distributed system executions. These graphs are unique in their value for distributed-system diagnosis and in their distinctly different characteristics compared to well-known social and web graphs. G^2 makes those graphs useful with new graph operators and with query support, and makes graph processing efficient with a distributed engine. By doing so, G^2 becomes an effective tool for distributed-system diagnosis and at the same time advances the state of art in distributed large-scale graph processing.

References

[1] The apache cassandra project. `http://cassandra.apache.org/`.

[2] Berkeley db. `http://www.oracle.com/database/berkeley-db/db/index.html`.

[3] The LINQ project. `http://msdn.microsoft.com/netframework/future/linq/`.

[4] Pagerank. `http://en.wikipedia.org/wiki/PageRank`.

[5] Phoenix compiler framework. `http://research.microsoft.com/phoenix/`.

[6] Program slicing. `http://en.wikipedia.org/wiki/Program_slicing`.

[7] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*. ACM, 2003.

[8] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, 2004.

[9] M. W. S. Bryan M. Cantrill and A. H. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, 2004.

[10] G. Candea, S. Bucur, and C. Zamfir. Automated Software Testing as a Service (TaaS). In *ACM SoCC*, 2010.

[11] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. In *OSDI*, 2006.

[13] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, O. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, 2002.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[16] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[17] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, 2007.

[18] R. Fonseca, M. J. Freedman, and G. Porter. Experiences with tracing causality in networked services. In *INM/WREN*, 2010.

[19] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[20] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *NSDI*, 2007.

[21] HADOOP. *http://hadoop.apache.org/*.

[22] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007.

[23] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^3S: Debugging deployed distributed systems. In *NSDI*, 2008.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, New York, NY, USA, 2010. ACM.

[25] A. J. Oliner, A. V. Kulkarni, and A. Aiken. Using correlated surprise to infer shared influence. In *DSN*, 2010.

[26] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[27] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.

[28] T. Wang and A. Roychoudhury. Hierarchical dynamic slicing. In *ISSTA*. ACM, 2007.

[29] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*. ACM, 2009.

[30] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*. USENIX Association, 2008.

[31] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*. ACM, 2010.

Context-based Online Configuration-Error Detection

Ding Yuan[1,2], Yinglian Xie[3], Rina Panigrahy[3], Junfeng Yang[4]*, Chad Verbowski[5], Arunvijay Kumar[5]*
[1]*University of Illinois at Urbana-Champaign,* [2]*University of California, San Diego*
[3]*Microsoft Research Silicon Valley,* [4]*Columbia University,* [5]*Microsoft Corporation*

Abstract

Software failures due to configuration errors are commonplace as computer systems continue to grow larger and more complex. Troubleshooting these configuration errors is a major administration cost, especially in server clusters where problems often go undetected without user interference.

This paper presents CODE–a tool that automatically detects software configuration errors. Our approach is based on identifying invariant configuration access rules that predict what access events follow what contexts. It requires no source code, application-specific semantics, or heavyweight program analysis. Using these rules, CODE can sift through a voluminous number of events and detect deviant program executions. This is in contrast to previous approaches that focus on only diagnosis. In our experiments, CODE successfully detected a real configuration error in one of our deployment machines, in addition to 20 user-reported errors that we reproduced in our test environment. When analyzing month-long event logs from both user desktops and production servers, CODE yielded a low false positive rate. The efficiency of CODE makes it feasible to be deployed as a practical management tool with low overhead.

1 Introduction

Software configuration errors impose a major cost on system administration. Configuration errors may result in security vulnerabilities, application crashes, severe disruptions in software functionality, unexpected changes in the UI, and incorrect program executions [7]. While several approaches have attempted to automate configuration error diagnosis [2, 20, 26, 29], they rely solely on manual efforts to detect the error symptoms [20, 26, 29]. As usual, manual approaches incur high overhead (e.g., requiring users to write error-detection scripts for each application) and are unreliable (e.g., security policy errors may show no user-visible symptoms). These drawbacks often lead to long delays between the occurrence and the detection of errors, causing unrecoverable damage to system states.

In this paper, we aim to automatically detect configuration errors that are triggered by changes in configuration data. These types of errors are commonplace and can be introduced in many ways, such as operator mistakes, software updates or even software bugs that corrupt configuration data. For example, a software update may turn off the "AutoComplete" option for a Web browser, which, as a result, can no longer remember usernames or passwords. An accidental menu click by a user may corrupt a configuration entry and cause an application toolbar to disappear. A seemingly benign user operation that disables the ActiveX control can unexpectedly disable the remote desktop application.

We consider configuration data because it captures important OS and application settings. Further, the data is typically accessed through well defined interfaces such as Windows Registries. We can thus treat the applications and the OS as black boxes, *transparently* intercepting and checking configuration accessing events (called *events* hereafter). This approach is lightweight: it does not require modifying the OS [13] or using virtual machines [29].

We focus on Windows, where applications use the Registry to store and access configuration data. In particular, we log all Registry events and analyze them online to automatically detect errors. While Windows has the largest OS market share [1] and is also the focus of many previous efforts [26], our methodologies can be generalized to other types of OS and configuration data.

Analyzing configuration-access events automatically for error detection faces three practical challenges. First, we need to efficiently process a huge number of events. A typical Windows machine has on average 200 thousand Registry entries [26], with 10^6 to 10^8 access events per day [23]. Commonly used learning techniques (e.g., [1, 28]) rarely scale to this level.

Second, we must automatically handle a large set of diverse applications. Different applications may have drastically different configuration access patterns. These patterns may evolve with user behavior changes or software updates.

Finally, our analysis must effectively detect errors without generating a large number of false positives. Configuration data is highly dynamic: there are, on average, 10^4 writes to Registry per day per machine, and 10^2 of them are writes to frequently accessed Registries

*This work was done when the authors were at Microsoft Research Silicon Valley.

[1]Specifically, Windows has 91% of client operating system market [22, 31] and 74% of server market [10].

that have never changed before. Application runtime behaviors such as user inputs, caching, and performance optimizations may all add noise and unpredictability to configuration states, making it difficult to distinguish between real errors and normal updates.

In this paper we present CODE, an automatic online configuration-error detection tool for system administrators. CODE is based on the observation that the seemingly unrelated events are actually dependent. The events externalize the control flow of a program and typically occur in predictable orders. Therefore, a sequence of events provides the *context* of a program's runtime behavior and often implies what follows. Further, the more frequently a group of events appear together, the more correlated they should be.

Thus, rather than analyzing each event in isolation, CODE extracts *repetitive, predictable* event sequences, and constructs invariant configuration access rules in the form of $context \rightarrow event$ that a program should follow. CODE then enforces these rules and reports out-of-context events as errors. By tracking sequences, CODE also enables richer error diagnosis than looking at each individual event. Once CODE detects an error, it also suggests a possible fix based on the context, the expected event, and the error event.

We implemented CODE as a stand-alone tool that runs continuously on a single desktop for error detection. It can also be extended to support centralized configuration management in data center environments. Our evaluation, using both real user desktops and production servers, shows that the context-based approach has four desirable features:

- *Application independent:* CODE requires no source code, application semantics, or heavyweight program analysis to generate contexts; it can automatically construct rules to represent more than 80% of events for most processes we studied.
- *Effective:* CODE successfully detected all reproduced real-world configuration errors and 96.6% of randomly injected errors in our experiments. CODE also detected a real configuration error on a coauthor's desktop.
- *Configurable false positive rate:* Since CODE reports only out-of-context events instead of new events, it will not report normal configuration changes as alarms. Further, the false positive rate is configurable. In our experiments it reports an average of 0.26 warning per desktop per day and 0.06 per server per day.
- *Low overhead:* CODE keeps only a small number of rules for detection and processes events as they arrive online. The CPU overhead is small (less than 1% over 99% of the time). The memory overhead is less than 0.5% for data-center servers with 16GB memory.

We explicitly designed CODE to detect *configuration* errors; our goal is *not* to catch all errors or malicious attacks. We view our focus on frequent event sequences as a good tradeoff. The high access frequencies indicate that errors in these events are more critical. Moreover, our detection takes place at the time when erroneous configurations are accessed and manifest. Hence, these errors are the ones that actually affected normal program executions, and CODE naturally concentrates on them.

This paper is organized as follows. We first discuss related work in Section 2 and introduce Windows Registry and a motivation example in Section 3. We then present an overview of CODE in Section 4. We next describe its rule learning (Section 5) and error detection (Section 6). We show our evaluation results in Section 7. Finally, we discuss our limitations and future work (Section 8) before we conclude (Section 9).

2 Related Work

To our best knowledge, CODE is the first automatic system for online configuration error detection. Below we discuss related work on configuration-error diagnosis and sequence-analysis based intrusion detection.

Configuration error diagnosis. Several diagnosis tools have been developed to assist administrators in diagnosing software failures. ConfAid [2] uses information-flow tracking to analyze the dependencies between the error symptoms and the configuration entries to identify root causes. Autobash [20] leverages OS-level speculative execution to causally track activities across different processes. Chronus [29] uses virtual machine checkpoints to maintain a history of the entire system states. KarDo [14] automatically applies the existing fix to a repeated configuration error by searching for a solution in a database. SherLog [33] uses static analysis to infer the execution path based on the runtime log messages to diagnose failures.

Another family of tools compares the configuration data in a problematic system with those in other systems to pinpoint the root cause of a failure [12, 26, 27]. They focus on the snapshots of configuration states, and use statistical tools to compare either historical snapshots or snapshots across machines. While it may seem feasible to extend these state-based approaches for error detection, our experiments showed that such approaches will generate a large number of false positives due to the noise in configuration states (e.g., constant state modifications or legitimate updates). In contrast, CODE reasons about *actions* rather than states for error detection.

The existing systems discussed so far have enhanced off-line diagnosis of configuration errors. However, they all require users or administrators to detect configuration errors. In contrast, CODE focuses on automatic error detection (it can further aid error diagnosis). The importance of having an automatic detection system is also recognized in [19]. Due to the complex dependencies

of modern computer systems, detecting faulty configuration states as early as possible helps to isolate the damage and localize the root cause of a failure, especially in server clusters or data centers with thousands of user-unmonitored machines.

Software resilience to configuration errors Candea et al. proposed a tool called ConfErr for measuring a system's resilience to configuration errors [4, 11]. ConfErr automatically generates configuration mistakes using human error models rooted in psychology and linguistics. ConfErr and CODE differ in their purposes. ConfErr can help improve software resilience to configuration errors and thus prevent errors from occurring, while CODE can be used to detect and diagnose configuration errors once they occur and is thus complementary.

Sequence analysis. A large number of intrusion detection systems (IDS) identify intrusions with abnormal system call sequences (e.g., [6, 9, 24, 32]). They

construct models of legal system call sequences by analyzing either the source code or the normal executions in an off-line learning phase. A deviation from the learned models is flagged as an intrusion.

By analyzing event sequences to identify predictable patterns, CODE shares similar benefits to run-time system-call analysis. However, our focus on configuration events instead of system calls leads to significantly different design decisions. Configuration access patterns constantly evolve, so off-line analysis used in IDS systems risks overfitting and producing outdated rules. Further, while IDS systems have to prevent sophisticated attacks [25] using conservative, non-deterministic models, CODE explicitly focuses on the potentially more critical frequent sequences using simple, deterministic rules. More importantly, the heavyweight learning algorithms that IDS systems commonly use make them difficult to scale to the volume of configuration access events, thus these systems are often unable to adapt to dynamic environments online. In contrast, the focus of identifying only invariant rules enables CODE to adopt and adapt much more efficient sequence-analysis methods to operate online.

Prior work (e.g., [8, 15]) has also used event transitions to build program behavior profiles. They mostly focus on depth-2 transitions on code call graphs. In contrast, CODE's event transition rules can consist of all possible lengths of prefixes, thus are more flexible and expressive when representing event sequences as contexts.

3 Background and A Motivating Example

In this section, we first introduce Windows Registry, the default configuration store for Windows applications. We then present a motivating configuration-error example and show how CODE can automatically detect and diagnose this error using contexts.

Key:	HKEY_LOCAL_MACHINE\Software\Perl
Value:	BinDir
Data:	C:\Perl\bin\perl.exe
Operation:	QueryValue
Status:	Success

Table 1: An example Windows Registry operation.

	Average	Maximum
Data modification	1051	5505
Key/Value creation	883	32676
Key/Value deletion	172	4997
Total	2106	43178

Table 2: Average and maximum number of Registry update operations/process/day (across 115 processes on a regular user desktop over one month period).

3.1 Windows Registry

Windows Registry is a centralized repository for software, hardware, and user settings on Windows machines. This repository makes it easy for different system components to share and track configurations.

Windows Registry is organized hierarchically, closely resembling a file system. Each Registry entry is uniquely identified by a Registry key and a Registry value. A Registry key resembles a directory and a Registry value a file name. A key may contain multiple subkeys and values. Given a key/value pair, Windows Registry maps it to Registry data, which resembles the content of a file. Hereafter, we will refer to Registry keys, Registry values, and Registry data as Keys, Values, and Data.

Table 1 shows a Windows Registry entry example. Its Key is a hierarchical path name with root Key HKEY_LOCAL_MACHINE, which stores settings generic to all users. The Key in the example stores settings about the Perl application. The Value/Data specifies that the Perl executable is located at C:\Perl\bin\perl.exe. Windows Registry supports about 30 operations (e.g., Createkey and OpenKey), each with a return value indicating the success or failure of the operation. Table 1 shows a successful QueryValue operation (given a Key/Value pair, fetch associated Data).

Previous studies have shown that a significant fraction of configuration errors are due to Windows Registry corruptions [7]. Software bugs, user mistakes, or application updates can all trigger unexpected Registry modifications that lead to software errors. In many cases, even a single entry corruption may result in serious application failures ranging from user-interface changes (e.g., a menu or icon missing) to software crashes.

While Windows Registry facilitates configuration access, it remains challenging to detect and diagnose configuration errors due to the complex and dynamic nature of Windows Registry. The number of Registry entries

1	Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate Op: OpenKey, Status: success, Value: "", Data: ""
2	Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate Op: QueryValue, Status: success, Value: WUServer, Data: http://sup-nam-nlb.redmond.corp.microsoft.com:80
3	Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate Op: QueryValue, Status: sucess, Value: WUStatusServer, Data: http://sup-nam-nlb.redmond.corp.microsoft.com:80
4-26	..., ... (check other settings)
27	Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU Op: QueryValue, Status:not exist, Value: DetectionFrequencyEnabled, Data:""
28 (normal)	**Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU Op: QueryValue, Status:not exist, Value: NoAutoUpdate, Data:""**
28 (error)	**Key: HKLM\Software\Policies\Microsoft\Windows\WindowsUpdate\AU Op: QueryValue, Status:0, Value: NoAutoUpdate, Data:1**
29-45 (normal)	..., ... (check other settings)

Figure 1: Registry access sequence of Windows update.

is huge—about 200K for an average machine, and this number is increasing [26]. Furthermore, Registry updates are highly frequent. As shown in Table 2, the number of updates can be as high as tens of thousands per process per day. Despite several recent proposals for automatic mis-configuration diagnosis, configuration-error detection remains an open problem.

3.2 A Motivating Example

In this example, we illustrates how CODE can detect and diagnose a real-world configuration error that disables the Windows automatic update feature (i.e., switch the OS to the manual update mode).

Given that Windows update often runs as a background task, users who normally leave automatic-update on will hardly notice that their computers have stopped checking for updates. Previous tools do not help in this case because they diagnose configuration errors only after users detect them. Consequently, this error may go undetected, leaving security vulnerabilities not patched and machines compromised. Early detection is thus critical to alert users to reset this important safety feature.

This configuration error was reported when a user removed a program that he or she thought was extraneous [30]. The program removal adds a Value "NoAutoUpdate" and Data 1 under the Key

K = [HKLM\Software\Policies\Microsoft\Windows\ WindowsUpdate\AU]

Since an average process can have over 2000 Registry modifications (i.e., writes) per day during its normal execution (Table 2), we need to determine which modifications are relevant to detection. One approach is to monitor and report modifications to only frequently accessed Keys. However, our experiments show that this approach would generate 154 false alarms per desktop/day, an unacceptably high number.

In our detection, CODE identifies that a rule involving a frequent sequence of exactly 45 Registry accesses is violated. By examining this sequence and its occurrence timestamps, we find that these events are issued by an `svchost.exe` process, which synchronizes with an update server and checks for available updates periodically (once per hour for Windows laptops). If there are updates available, the checking process will proceed to download and install the updates.

Figure 1 shows this 45-event sequence. It begins with an OpenKey operation on registry Key "HKLM\...\WindowsUpdate", which stores all the information about Windows update. Next, `svchost.exe` accesses this Key and all its Values. For example, the second and third operations show that `svchost.exe` queries the URLs of the windows update server and the status reporting server.

At the 28th event, `svchost.exe` queries the Value "NoAutoUpdate" (highlighted in Figure 1). Since this Value does not exist during normal execution, the QueryValue operation will return "Value not found" and `svchost.exe` continues to check other automatic update options. However, after the Value "NoAutoUpdate" is created with Data 1, the operation returns "Success", causing `svchost.exe` to prematurely stop without further checks.

Since the 45-event sequence occurs frequently in normal execution, CODE will learn a set of rules from this sequence. In particular, it will identify the first 27 events as the context for the 28th event. Thus, in the error case, CODE successfully detects the deviation. In addition, CODE knows (1) the context, the expected event and the event actually happened and (2) what process and the time at which the process created the problematic Value. It can thus pinpoint the root cause and recover the error.

4 System Overview

From a high level, our approach identifies *predictable* configuration-access rules from program executions for error detection and diagnosis. From the example described in Section 3, we see that each Windows update check triggers a sequence of 45 Registry accesses. This entire sequence is deterministic and thus predictable. This behavior is not surprising, as configuration-access patterns are usually reflective of a program's control flow. When a program runs the same code blocks with same/similar user inputs, the set of external events tend to be same/similar and in order.

We focus on only these predictable event sequences in our detection. For each event in such a sequence, its preceding event subsequence provides the *context* for the current program execution point. A deviation from the predicable event sequence suggests that the corresponding program's control flow might have changed, which

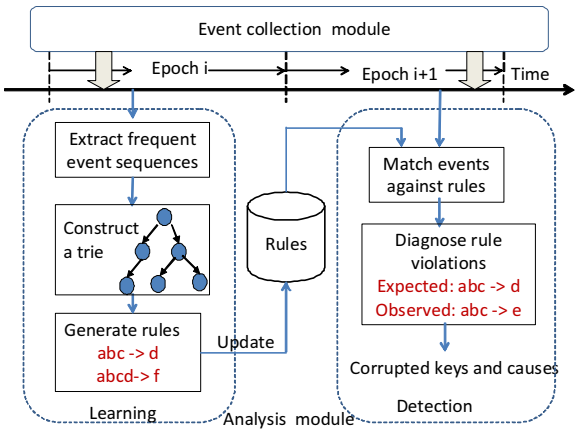


Figure 2: The CODE system architecture.

may indicate the existence of configuration errors. In this case, the expected sequence and the actually observed one are further used to diagnose the error's root cause.

However, not all configuration-access events are predictable. A program's runtime behavior such as caching, optimizations, or the use of temporary files may all affect the program's control flow. Correspondingly, accessing the configuration data will be less predictable. We may observe a large number of temporary events, and even the same set of events may exhibit completely different timing orders. The challenge is how to differentiate the two cases and identify only predictable patterns from a voluminous number of events.

Given the complexity and dynamics of Windows Registry, CODE must meet the following two requirements to realize online detection and diagnosis:

- **Efficient:** The tool should have low timing complexity in order to process events as they arrive in real time. The number of Registry events to process is on the order of 10^6 to 10^8 per machine per day.
- **Effective:** As an online tool, CODE needs to distinguish true errors from volatile or benign changes.

We implement CODE as a stand alone tool, monitoring each host independently (Section 8 discusses our deployment of CODE as a centralized manager for data centers). We structure CODE into two parts: an event collection module and an analysis module (Figure 2). Both run simultaneously as a pipeline. The collection module writes Registry operations to disk and the analysis module reads them back for learning, detection, and diagnosis. We chose this architecture to keep the collection module simple; otherwise, it may perturb the monitored processes. We chose files as the communication method between the two modules (instead of sockets) for flexible control over analysis frequency (e.g., every few seconds to minutes).

The core of the event collection module is a Windows kernel module written in C++, similar to FDR [23]. It intercepts all Registry operations and stores them in a buffer in highly compressed forms. It then writes them to disk periodically.[2] Each event contains the following fields: event time-stamp, program name represented by the entire file system path to the executable, command line arguments, process ID, thread ID, Registry Key, Value, Data, operation type (e.g., OpenKey and QueryValue) and operation status.

The analysis module is implemented in C#. It includes a learning component and a detection/diagnosis component. Both learning and detection are done by analyzing the event sequences at a per-thread level because they faithfully follow the program's control flow in execution. For compact representation, we fingerprint a Registry event to generate a Rabin hash [18] by considering all of its fields excluding the time-stamp, the process ID, and the thread ID.

$$\underbrace{a\ b\ c}_{\text{context}} \longrightarrow d$$

Figure 3: Example of a rule.

The learning component takes the Registry event sequences as input, and generates a set of *event transition rules*. Figure 3 shows an example rule. In this example, a, b, and c each represents a unique Registry event. This rule means if we have observed events a, b, and c in sequence, then the next event is determined to be d. In other words, event sequence *abc* is the context of event *d* if and only if *abc* will be always followed by *d* with no exceptions (We do not consider non-deterministic cases where *abc* can be followed by other events such as *e*, as majority of the important errors can be captured by deterministic cases in our experience).

We further require the number of occurrences of the rule sequence to exceed a certain threshold for it to be deemed as a rule. We use the complete command line that launched a process to group the set of threads sharing the common executable name and input arguments[3]. The frequency of a rule is thus measured over all the event sequences across a process group. Finally, the set of learned rules are updated periodically by *epochs* and stored in a rule repository as illustrated in Figure 2. We define an *epoch* as a time period where we observe a fixed number of events, so that the rules learned from one epoch can be applied immediately in the next epoch.

The detection component takes the set of learned rules and applies them to detect errors as new events arrive.

[2]The overhead is negligible, even when flushing the buffer every minute [23].

[3]Different arguments often lead to different program execution paths for different tasks. Applications that launch at machine boot time often start with fixed arguments. In Windows, many applications have a graphical icon on the desktop that launches the application with fixed arguments each time.

n case of a rule violation, the detection module performs a set of checks to facilitate diagnosis based on the ule sequence, the expected event, the Registry write that aused the error, and the actually observed event. In the ext two sections, we describe the details of rule learning nd error detection/diagnosis.

5 Learning Configuration Access Rules

his section describes how CODE generates event transiion rules from input Registry-access sequences. These nput sequences consist of registry accesses at thread granularity for each process group (i.e., all processes that hare the same executable name and command-line argunents). Figure 2 shows the three steps of this procedure: 1) generate frequent event sequences, (2) construct a trie i.e., a prefix tree) to represent the event transition states, nd (3) derive invariant event transition rules based on he trie. For efficient detection, CODE represents the set f output rules in the form of a trie with labeled edges, nd each process group has a separate trie.

Throughout the process, CODE has time complexity inear in the number of events processed. Although CODE generates a set of frequent event sequences independently from each epoch, meaning that a sequence as to appear frequently enough within one epoch to be earned by CODE, it maintains the labeled tries in memry across epoches and updates them incrementally. We vill show in Section 7.3 that the generated trie sizes are mall for most of the programs.

5.1 Frequent Sequence Generation

he first step of generating frequent sequences is the nost critical, since it provides the candidate event sets or generating rules as well as the potential context engths. To identify frequent event sequences, one option is to generate hash values for fixed-length event subequences, and then count their frequencies. We may potentially leverage data structures such as bloom filters [3] o optimize space usage. However, this option is not desirable because it is difficult to pre-determine the event ubsequence lengths. Although we may choose several opular lengths (e.g., 2, 4, 8), the semantically meaningul event sequences can be very large (as illustrated in ection 3) and can have varied lengths. Popular techniques such as suffix trees [16] are not applicable either. hey typically require the entire input sequence to be vailable. Furthermore, their space-time requirements re not efficient enough to deal with a large number of Registry events arriving in real time.

In order to generate the longest applicable frequent ubsequences efficiently, CODE adopts the Sequitur [17] lgorithm. Given a sequence of symbols, Sequitur identifies repeated sequence patterns and generates a set of grammar rules to hierarchically represent the original sequence. Figure 4 shows an example input sequence and the hierarchical grammar rules derived by Sequitur. The lower case letters represent the input symbols, and we use upper case letters to denote the derived symbols.

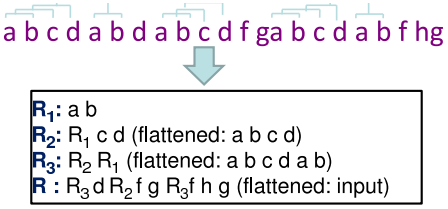


Figure 4: An example of Sequitur hierarchical rules. We also show the flattened rule in the parenthesis.

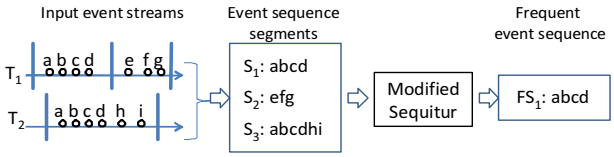


Figure 5: Generating frequent event sequences. T_1 and T_2 are two threads belonging to the same process group.

During learning, the default epoch size is 500K events, which can span from hours to days for different processes.[4] For each epoch, CODE does not need to store the complete input sequence because the hierarchical representation makes the original sequence more compact. In practice, the number of symbols to store in memory is roughly on the order of the number of distinct Registry events, which is around only 1% of the total events [23].

Compared with other methods, Sequitur has a linear time complexity and reads only one pass of data in streaming mode. Although it may generate sub-optimal frequent sequences, we found it acceptable in our application, as low time complexity is an important requirement. To apply Sequitur in our context, we make the following two modifications to the algorithm:

Analyzing multiple sequences simultaneously. The incoming events processed by CODE contain not a single event sequence, but multiple sequences. These sequences come from different processes and different threads in the same process group. In addition, we observe that events belonging to the same task often occur in a bursty manner. Mixing events from these semantically different tasks as one sequence would create unnecessary noise. We thus segment them into per-thread per-burst sequences (the default time interval between two bursts is one second), as shown in Figure 5.

The original Sequitur algorithm, however, analyzes only one sequence at a time. We thus modify it to take multiple sequences. We could maintain a separate gram-

[4] After learning, the detection takes place in real time.

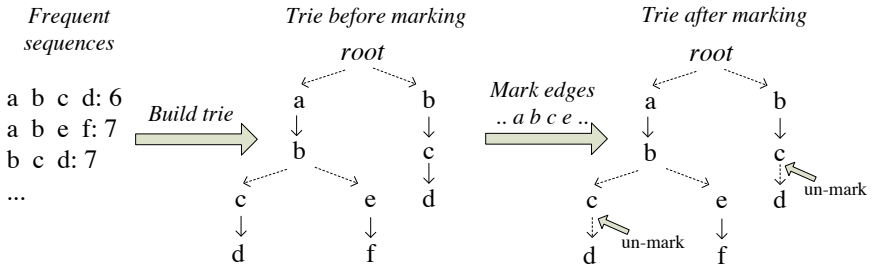


Figure 6: Constructing a trie from frequent event sequences and identifying its rule edges.

mar table (needed for Sequitur) for each sequence, but this approach would miss common subsequences shared across different threads in the same group. For example, in Figure 5, both threads share the subsequence `abcd`. Thus a grammar table is shared among all sequences. This sharing also reduces CODE's memory usage.

With grammar table sharing, one complication arises when a sequence S_x completely contains another one S_y. To avoid storing the same sequence twice, Sequitur would replace the redundant copy of S_y in S_x with a pointer to S_y. However, we cannot expand S_y if new events come in, because this expansion may make S_y no longer a subsequence of S_x. To solve this problem, we give S_y a fresh name S'_y each time we expand it.

Flattening the hierarchy: The second modification is to flatten the default hierarchical symbols output by Sequitur to event symbols in order to construct the trie later (illustrated by Figure 4). To ensure each learned sequence is not too short, we select a flattened event sequence only if its length is above a pre-defined length threshold l (by default $l = 4$) and its sequence is above a pre-defined frequency threshold s (by default $s = 5$). We call the frequency of an event sequence as its *support*.

Although the rule flattening process is relatively straightforward, correctly computing the support (i.e., frequency) of the expanded sequences is a more involved task. In Figure 4, R_1 appears at both R_2 and R_3, and R_2 further appears at R_3. CODE takes a top-down approach to traverse the hierarchical representations for computing the correct support. The final output of this step is a set of frequent event sequences with support greater than s.

5.2 Event Trie Construction

After CODE generates the frequent sequences from input events, it proceeds to construct an event trie in the form of a prefix tree to store all the frequent sequences from all threads of each process group. Figure 6 shows the construction of an example trie. In a trie, each node represents a Registry access event (encoded as a Rabin hash), and each directed edge represents the transition between the two corresponding events in temporal order.

The adoption of a trie representation serves a couple of important purposes. First, it represents the temporal transition relationships between different events, providing the basis for deriving event transition rules. Second, we found that many frequent event sequences have common prefixes. Hence a prefix tree explicitly encodes the divergence of different event paths from a single point.

We further optimize the trie data structure to make it more compact. An observation is that many event sequences share suffixes as well. In practice, merging common suffixes is very effective in reducing the trie size (by half). Meanwhile, this optimization still preserves the event transition relationship and ensures the correctness of the derived rules.

5.3 Rule Derivation

With a trie, CODE proceeds to derive *event transition rules* that all threads from the same process group have to follow. We look at only the rules that were never violated. Our approach is to identify those event transitions $a \rightarrow b$ that are deterministic given the sequence of events from the root to a. We define such an edge as a *rule edge*. Clearly, only edges from nodes with only one outgoing edge are rule edge candidates.

However, simply counting outgoing edges is incomplete. For example, given a frequent sequence $abcd$, we can construct a trie of 4 nodes, and the edge from $c \rightarrow d$ appears to be a rule edge. However, there may exist a sequence $abce$ that did not occur frequently enough to be selected as a popular sequence. In this case, the transition $c \rightarrow d$ is not deterministic.

For each newly created rule edge, CODE determines whether it is truly a deterministic transition by checking it against the upcoming event sequences in the next epoch. Figure 6 shows this edge-marking process. Doing so defers the use of this edge for detection. It is worth noting that for each event, CODE identifies all possible matches based on the preceding subsequences. Additionally, CODE also starts from the root every time to capture subsequences that begin with the current event. During the edge-marking process in Figure 6, if the incoming event e is following sub-sequence abc, we will un-mark the two $c \rightarrow d$ transitions from rule edge in the trie.

6 Error Detection and Diagnosis

This section describes how CODE detects configuration errors using the learned rules and further outputs diagnosis information. Since the labeled trie structure captures the rules as deterministic event transitions and is efficient at matching sequences, we conveniently reuse this data structure for error detection without explicitly representing the rules. The detection algorithm is thus simple and similar to the edge-marking process in Figure 6, except when we see a violation, we report a warning rather than un-marking the transition. This online detection method ensures that we can detect a configuration error as early as possible, before it affects other system states.

6.1 False Positive Suppression

In the rule-learning process, the support threshold s can be used to configure the false positive rate. A larger s usually implies a smaller false positive rate, but we may also miss some real errors. We further evaluate this parameter in Section 7.2.

Additionally, we use three techniques to reduce CODE's false positive rate. First, before CODE reports a warning, it performs an additional check to ensure that the violated (i.e., expected) event does not appear in the near future. So if $abc \rightarrow e$ is a rule that is violated by observing abc followed by f, then we monitor the events for a delay buffer (set to 1 sec) to check if e appears; if it does, we suppress the warning. The idea behind this check is that since we are looking for corruptions of Registry Keys/Values, if f is indeed a corruption of the Key/Value corresponding to the Registry in event e, then e should not appear again. Otherwise it is perhaps simply a benign program flow change.

Second, if multiple alarms are generated in a 1 second delay buffer, CODE only reports the first one as the others are likely manifestations of the same root cause. We found the first alarm is always the true root cause in our experiments (see Section 7.1.1).

The third technique is *cooperative false positive suppression*: aggregate warnings from all machines, and report only unique ones. We consider two warnings identical if they warn about the same Key, Value, and Data. We canonicalized user names when comparing Registry Keys (More canonicalization would help, but it is beyond the scope of this paper). This technique effectively reduced the number of false positives by 30% in our experiments, though it can be turned off for privacy concerns.

6.2 Error Diagnosis

CODE also provides rich diagnosis information after error detection. When a process violates a rule $context \rightarrow event$, CODE knows precisely the context, the expected event, the violating event, and the violating process. Such information can help diagnosis in a few ways.

First, CODE allows the operator to understand how the Registry in the expected event was changed by tracking which process, at what time, modified the entry that caused the error. To do so, CODE uses a modification cache to store the last modification operations (along with timestamps) on the Registries in the rules. Because the rules track only frequently accessed Registries and the majority of the accesses to these Registries are read-only events, we need only a small cache. In practice, the size of the modification cache is always smaller than 2,000 events for all the machines that we used in our experiments. The typical size of 200 events is enough for the majority of them.

Second, the expected event and its context often provide enough information regarding the program's anomalous behavior to the administrator. They also provide the candidate Registry entries for recovery. In the "auto-update error" example in Section 3.2, the expected event has empty Data for Value `NoAutoUpdate`, while the violating event has "1" as the Data. Further the expected event belongs to a sequence where svchost.exe is checking for auto-update setting. Such information provides hints to the administrators about the root causes.

Finally, CODE returns all the processes whose rule repositories involve the corrupted Registry. Operators can use this information to examine whether the same configuration error might affect other programs.

7 Evaluation

We deployed CODE on 10 actively used user desktops and 8 production servers. In our month-long deployment, we set the data collection interval to every one hour. We ran the analysis module separately off-line on the collected registry-event logs. This allowed us to conveniently examine the logs in detail. For the off-line analysis, it took about 12 hours to process each machine's one-month log. We also evaluated the same version of CODE using one minute intervals to measure its online analysis performance.

To demonstrate the value of using context, we also implemented a *state-based approach* that does not use context for error detection and compared it with CODE. Instead of looking at sequences, this approach tracks commonly used Registry Key/Value entries and raises an alarm if the Data field has not been observed before. To ensure a fair comparison, we applied the same parameters used by CODE as well as the set of false positive suppression heuristics described in Section 6 whenever applicable. Below we present our evaluation results.

7.1 Detection Rate and Coverage

We first evaluate CODE using real-world configuration errors and randomly injected Key corruptions.

Error name	Description
Doubleclick	When double clicking any folder in explorer, "Search Result" window pops up.
Advanced	IE advanced options missing from menu.
IE Search	Search dialog will always be on the left panel of IE that can't be closed.
Brandbitmap	The animated IE logo disappears.
Title	IE title changed to some arbitrary strings.
Explorer Policy	Windows start menu becomes blank.
Shortcut	In explorer, clicking the shortcut to a file no longer works.
Password	IE can no longer remember the user's password.
IE Offline	IE would launch in offline mode and user's homepage can't be displayed.
Outlook trash	Outlook asks to permanently delete items in the "Deleted Items" folder every time it exits.

Table 3: Description of the 10 reproduced errors.

7.1.1 Detection of Real-Errors

The real world error discovered by CODE was caused by Hotbar Adware [21], which unexpectedly infected one co-author's desktop. This adware adds graphical skins to Internet Explorer (IE), and modifies a group of Registries related to the Key "HKLM\Software\Classes\Mime\Database\Content type\App". CODE successfully detected rule violations at the IE start-up time. CODE further provided diagnostic information to help remove the IE tool bars created by the adware.

Additionally, we manually reproduced 20 real-user reported errors to evaluate CODE. These errors were selected from a system-admin support database. The only criteria we used in our selection was whether these errors were triggered by modifications to Windows Registry and were reproducible.[5] The error reproduction process exactly followed the set of user actions that triggered the software failures as described in the failure report. The 20 errors involved nine different programs, including popular ones such as Internet Explorer, Windows Explorer, Outlook, Firefox.

CODE successfully detected all these reproduced errors. Due to space constraints, we do not describe all of them, but list the 10 representative ones in Table 3. To further evaluate the effectiveness of CODE across different environments, we reproduced these 10 errors in 5 different OS environments (one of them was a virtual machine). Not all of these 10 errors can be reproduced on all 5 machines; out of all combinations, we were able to reproduce 41 cases.

Among these 41 cases, CODE detected 40 cases and missed only 1 case (Table 4). Further investigation on the missing case showed CODE had over-fitted the context for that error; that is, the context learned was longer than that observed after the reproduction. We suspect there might exist two different program flows that preceded the access to the corresponding Registry Key, and CODE learned a longer context than what was observed during detection.

[5]Some errors require special hardware setup or specific software versions to reproduce.

Machine OS and IE version	Server 03 IE 6	Vista IE 7	xp-sp2 IE7	xp-sp3 IE 7	xp-VM IE 6
Doubleclick	1 (1)	1 (1)	1 (3)	1 (3)	1 (2)
Advanced	1 (1)	1 (1)	1 (2)	1 (1)	1 (6)
IE Search	1 (10)	N/A	N/A	N/A	1 (7)
Brandbitmap	N/A	N/A	N/A	N/A	1 (3)
Title	1 (1)	1 (1)	1 (2)	1 (3)	1 (3)
Explorer Policy	1 (1)	1 (2)	1 (2)	1 (5)	1 (2)
Shortcut	1 (1)	1 (1)	1 (3)	1 (1)	1 (2)
Password	N/A	1 (2)	1 (1)	1 (2)	1 (2)
IE Offline	1 (1)	1 (1)	1 (2)	-	1 (1)
Outlook Trash	1 (2)	1 (2)	1 (2)	1 (2)	N/A

Table 4: Detection results of reproduced real errors. The first number in each box is the rank of the root cause event, and the second number in the parenthesis is the total number of violations observed in detection. N/A means we couldn't reproduce that error on that machine, and "-" is the case CODE missed.

Table 4 lists the total number of violations before CODE aggregated the warnings within the one second delay buffer. In all these cases, the root cause event was the first event that occurred. The other violations all happened in a burst right after the first one. By aggregating warnings (Sect. 6.1), only the first alarm is reported.

Indeed, manual inspection suggests those additional violations are not false positives but are highly correlated to the root cause. For example, the `Outlook Trash` error is triggered by modifying the Data of Key "HKCU\Software\Microsoft\Office\11.0\Outlook\Preferences\Emptytrash" to 1. This error caused an alert window to pop up on each exit of Outlook, asking whether to permanently delete all items in the "Deleted Items" folder. This alert window is related to another Registry Key "\HKCU\Software\Microsoft\Office\11.0\Outlook\Common\Alerts", whose settings were changed during the error, causing CODE to report additional violations.

Based on the diagnosis information output by CODE, we can easily recover all the reproduced errors by changing the corrupted Registry entries back to the expected ones. However, due to the complex dependencies between today's system components, we expect automatic recovery to be a challenging topic for future work.

7.1.2 Exhaustive Key Corruption

To evaluate the coverage of CODE's error detection, we manually deleted every Registry Key that is frequently accessed ($\geq$ 2 times) by a process on a virtual machine. Note that this does not imply CODE can detect configuration errors caused by only Registry deletions. Any change to Registries such as modifications or new Key/Value creations, can be detected by CODE so long as a future access to these modified Registries violates a learned rule. For example, the AutoUpdate error in Section 3.2 was caused by modification to a Registry Data.

The process we chose is Internet Explorer (IE), which has both the maximum number of Registries and distinct Registry Key accesses on a typical desktop machine. We ran a program that simulates user browsing activities by periodically launching an IE browser, visiting a Web site, and then closing the browser. After running this program for two hours (for the learning phase), we deleted every Registry Key that IE accessed more than twice during the two hours, one at at time. After each corruption, we ran the program twice that simulates a user's Web visit and let CODE perform detection. We then recovered the corrupted Key before proceeding to the next Key corruption.

Total Registry accesses	Registry writes	Distinct Keys
2,097,642	275,549 (13.1%)	1,247
Frequent Registry Keys ($\geq$ 2 times)	Successfully corrupted Keys	CODE detected corruptions
783 (62.8%)	387	374 (96.6%)

Table 5: Summary of the Key corruption experiment.

Table 5 summarizes the statistics and the results. Among the 387 successfully corrupted Keys, CODE detected 374 (96.6%) of them. Note not every frequently accessed Key can be corrupted. Among 783 of the frequent Keys, we successfully found and corrupted only 387 of them. The remaining Keys were temporary to the life time of a particular IE instance. Since our experiment periodically launched a new IE instance, those temporary Keys no longer existed at the deletion time.

In total, CODE failed to detect 13 of the corrupted Keys, among which, 12 are Keys or sub-Keys of the following 4 Keys:

- HKEY_LOCAL_MACHINE\software\ classes\rlogin
- HKEY_LOCAL_MACHINE\software\ classes\telnet
- HKEY_LOCAL_MACHINE\software\ classes\tn3270
- HKEY_LOCAL_MACHINE\software\ classes\mailto

These Keys store settings about the dynamically linked libraries for handling four application-layer protocols and they are periodically queried by IE. During the exhaustive Key-corruption experiment, we deleted a Registry Key "AutoProxyTypes" that stores settings about automatic Internet sign-up and proxy detection. The deletion of this Key may have triggered persistent program behavior changes in IE, which switched to an alternative configuration option that did not rely on the above four Keys to perform Internet sign-up and proxy detection. This example also suggests that recovering from errors triggered by configuration changes may require more than reversing these modifications.

Total Frequent Key Access	Frequent Access CODE Captures	Distinct Accesses To Frequent Keys
2,090,777	2,083,912 (99.7%)	2,400
Distinct Access CODE captures	Accesses with single context	Average Number of contexts
2,400 (100.0%)	1,743,708 (83.4%)	1.74

Table 6: Event context statistics.

To further understand the predictability of using contexts for detection, we measure the number of the Registry accesses that fall into contexts, where our detection is applicable. Table 6 shows that out of the total 2,090,777 accesses to the frequent Keys, 2,083,912 (99.7%) of them fall into some contexts, and thus may be captured by CODE. Furthermore, 83.4% of the frequent accesses belong to a single non-overlapping context. This means that their access happened in only one deterministic way. On average, for each frequent Registry access, it has 1.74 contexts. For those Registry accesses that have more than one context, most of them are related to the settings of dynamically linked modules that may be shared by different components in IE, resulting in more than one context.

7.2 False Positive Rate

We evaluated the false positive rate of CODE using month-long Registry access logs from the following two sets of machines: (1) 8 production servers with similar hardware and workloads and (2) 10 desktops used by two interns, four researchers, one research lab manager, and three part-time vendors, giving us a diverse set of workloads. Other than the Hotbar Adware, we were unaware of any other configuration errors reported for the log-collection time period.

	CODE			State-based
Num/day/machine	Average	Max	Min	Average
Server	0.06	0.27	0	13.67
Desktop	0.26	0.96	0	153.83

Table 7: Summary of false positive rates (in terms of the number of warnings/machine/day) across 10 desktops and 8 servers.

Table 7 shows the false positive rates of CODE. Over the 30 day period with hundreds of billions of events from all machines, CODE reported a total of 78 warnings with an average of 0.26 warning/desktop/day and 0.06 warning/server/day. As a comparison, the state-based approach reported three orders of magnitude more, on

Name	Description	Percentage
File Association	The default program used to open different file types is changed.	24.1%
MRU List	Changes to most recently accessed files tracked by applications (e.g., explorer and IE)	12.7%
IE Cache	The meta-data for the IE Cache entities is changed.	3.8%
Session	The statistics for a user login session are updated.	3.8%
Environment	Environment variable changes.	2.5%

Table 8: Top 5 reasons for causing false positives on one machine. The "Percentage" column shows, using the 5 categories, the percentage of alarms that can be summarized over all alarms from all machines.

average 153.83 warnings/desktop/day and 13.67 warnings/server/day. This difference can be explained by several reasons. First, many modifications to frequently accessed Registries do not occur in any frequent sequences (i.e., no context). Second, multiple Registry modifications often belong to a single sequence where CODE reports only the first modification as a warning while the state-based the approach reports all of them. Finally, some modified Registries will never be accessed again after the modification. While the state-based approach reports all such cases as warnings, CODE does not because it reports a warning only when the modified Registry is read again.

We further examine the time distribution of the warnings generate by CODE. Figure 7 shows that for the desktop that generated the largest number of warnings (0.96/machine/day in Table 7), only 4 processes reported a total of 29 warnings during the 740 hours (more than 30 days). Most warnings are clustered in time, and are likely caused by the same configuration modification event.

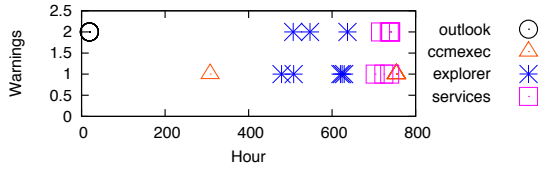


Figure 7: Number of warnings per hour generated by the desktop that had the most number of warnings.

We analyzed the different causes of the false positives on user desktops and found that they can be categorized into a few types (Table 8 summarizes the top five causes). Some of them (File Association and Environment Variable) are intended configuration changes issued by users; the others (Most Recently Used List, IE Cache, and Session Information) are temporary-data changes. By using regular expressions to filter the Registry Keys that fall into these top five causes, we can potentially reduce the false positive rate to 0.14 warnings/desktop/day.

We also observed a significant overlap in the false positives generated across different machines. Without the cooperative false positive suppression heuristic that merges false positives across machines, the false positive rate in an isolated detection would have increased from 0.26 to 0.36 warnings/desktop/day.

7.2.1 Analysis Sensitivity

We study CODE's sensitivity to workload and the support threshold (i.e., the number of occurrences for a frequent event sequence to be learned as a rule) in this section.

Workload sensitivity. Table 7 shows that CODE's false positive rate is four times lower on servers than on user desktops. This is because server workloads are less interactive, and thus, their Registry access logs are less noisy. To evaluate the workload sensitivity, we measure the false positive rate of different programs for all the machines in our experiment. Among all the programs running on the servers, only 2 ever reported warnings; for programs running on desktops, 12 reported warnings. The program Windows Explorer (`explorer.exe`) generated the maximum number of warnings, contributing to 1/3 of the total alarms followed by Internet Explorer (`iexplore.exe`) and Windows Login (`winlogon.exe`). Windows Explorer is like the Unix shell for Windows and is highly interactive. While CODE currently uses the same support threshold 5 for learning frequent sequences, we can adjust the false positive rate by setting a larger support threshold.

Support-threshold sensitivity. As discussed above, an important parameter is the support threshold for separating frequent and infrequent sequences. We evaluated this sensitivity using the desktop with the highest false positive rate (0.96/machine/day in Table 7). Figure 8 shows the result. As was expected, using a larger threshold decreased the false positive rate. Users and administrators can tune this parameter to trade-off detection rate vs. false positive rate.

7.2.2 Impact of Software Updates

Software updates are frequent on modern computers. Their activities may be intrusive and change a program's configuration-access patterns. We study the impact of software updates on the false positive rate in this section.

We used the logs collected from the 10 desktop machines for our analysis. We treat a warning as a software-update related false positive if the corresponding Registry was last modified by one of the Windows software update processes (e.g., `ccmexec.exe`, `svchost.exe`, `update.exe`) and Windows software installation processes (e.g., `msiexec.exe`).

Among the 78 false positives reported by CODE, only 5 were due to software updates, averaging to 0.017 warn-

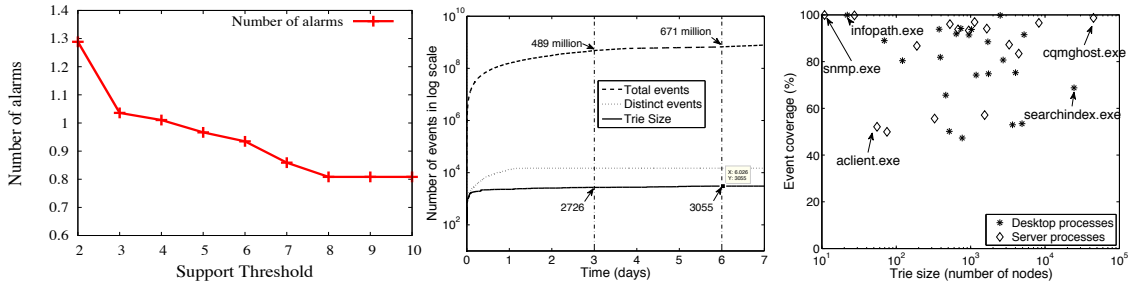


Figure 8: (a) Sensitivity of false positive rate vs. support threshold. (b) The growth of the trie size and the number of events over time for IE (desktop) in log scale. (c) Trie size vs. event coverage for different processes on two machines.

ing per desktop/day or 0.139 per update across total 36 updates from these machines. These 5 warnings were caused by two environment variable updates, one display icon update, one DLL update, and one daylight saving start date update. This small false-positive number is not surprising, as software updates tend to fix bugs and add new functionalities, but do not change the existing frequent configuration-access patterns.

CODE learned the new access patterns introduced by software updates as new rules, rather than considering them as false positives. For example, after a large Office update on a desktop, the trie size of the corresponding program increased by 10% within one day. Otherwise, the trie size was relatively stable.

We further examine the most intrusive update we found in the logs: an update from Office Service Pack 2 to Service Pack 3 [5]. This upgrade includes more than 200 patches. It affected 7 of the Office applications, created and modified more than 20,000 keys, but caused only one false positive warning. A closer look revealed that while this update created many keys, the majority of them were temporary keys for bookkeeping and were deleted right after the update, causing no warnings. This update additionally modified or deleted 61 existing keys; only 10 keys overlapped with the rules CODE learned and they were all captured in one rule, causing the only warning. These 10 keys specified the daylight saving start dates of 10 countries and were frequently queried by Outlook [6], resulting in a CODE rule. When the Office update changed these keys, CODE detected a rule violation.

7.3 Performance Evaluation

When we deploy CODE in online mode, where it periodically (every minute) processes Registry events arriving in real time, the CPU overhead is very small–less than 1% over 99% of the time, with a peak usage between 10%-25% (on an AMD 2.41GHz due-core machine). The current memory usage is between 500 MB-900 MB.

[6]Outlook queries these keys to determine how to display the calendar items based on the current time zone.

The memory overhead is largely caused by maintaining sets of tries, one for each process group. Figure 8 (b) plots the trie size growth over time in log scale for an IE process. The trie size is about 2000-3000 and converges roughly after 1 day. In contrast, the number of Registry events can be up to hundreds of millions. Even the number of distinct events is one order magnitude larger than the trie size, suggesting CODE is effective in reducing the event complexity.

We proceed to examine the trie sizes for different processes in Figure 8 (c). For the majority of the processes, their trie sizes are consistently small, on the order of hundreds to tens of thousands of events. The total trie size across all processes on a machine is still small, on average 529,500 per user desktop and 97,042 per server. Given each trie node requires around 12 bytes (8 byte Rabin hash + 4 byte pointer), maintaining all the tries requires around 1MB-6MB in the ideal, optimized case. We suspect a large portion of the current memory overhead is caused by both caching the event sequences during the learning phase and the C# overhead. Such overhead can be potentially reduced by using sampled epoches to reduce the learning frequency, and by re-implementing the analysis module in C++.

Figure 8 (c) also shows the percentage of unique events included in the tries defined as *event coverage*. This metric roughly tracks the Registry-access predictability. We found that most of the processes have over 80% of event coverage. In particular, the `snmp.exe` process running on the server is highly predictable, where a trie with 27 unique events can represent 99.77% of all its Registry access events.

One of our goals is to use CODE to monitor server clusters or data center machines for detecting abnormal configuration changes. A typical server cluster consists of machines with similar hardware, software settings, running similar workloads. In this scenario, CODE could offload the analysis task from each server to a small number of centralized management servers.

We run CODE in a centralized mode, constructing a single centralized trie that consists of all the rules from

	Trie Size (%)	Memory MB (%)
1 machine	98,042	503
2 machines	119,503 (21.9%)	510 (1.4%)
4 machines	134,892 (12.9%)	560 (9.8%)
8 machines	139,918 (3.7%)	600 (7.1%)

Table 9: The size and memory usage of a centralized trie constructed by analyzing events from multiple machines. The trie size is monitored after 3 days, and the memory usage is the average usage in one day.

multiple machines. Table 9 shows the growth of the trie size and the memory usage as we increase the number of machines to monitor. As we see, the trie size grows by only 3.7% when the number of machines to monitor increases from 4 to 8. This suggests that rules learned from multiple machines can be applied to other similarly configured machines (i.e., with similar hardware, software and workload). For centralized configuration-error detection, the memory overhead is on average about 0.4% per machine for 16GB-memory servers. We leave it as future work to fully generalize the CODE approach to perform centralized data-center management.

8 Discussion

Limitations: Not all configuration errors can be detected by CODE. By focusing on changes to configuration data and their access patterns, CODE may not detect errors introduced at system or software installation/setup time. To detect these errors, we can extend CODE to process event sequences across machines, so that errors on one machine can be detected by comparing Registry event sequences from another properly installed machine. Previous work [26] has also showed encouraging results by cross referencing static configuration states in a similar way. If a configuration error is caused by an event without any context, CODE cannot detect it either. However, in our evaluation, we have not encountered such errors.

We have evaluated CODE on only Windows Registry, but we believe CODE's underlying techniques can potentially be generalized to other configuration formats, such as Unix's configuration files under /etc/. However, in Unix, different applications manage their own configuration data in their own format, so it might require per-application instrumentation to collect the configuration data access trace.

CODE can be deployed as both a stand-alone tool running on end user's desktops and a centralized management tool used by system administrators to monitor multiple machines in a data-center or a corporate network. We expect CODE to work better in the latter scenario for the following reasons. First, end users might have no clue on how to deal with warnings for filtering false positives. Second, with centralized management, an end user desktop can be spared from the 500-900MB memory overhead (the event collection component still needs to run on end user machines, but it has a negligible overhead [23]). Third, our cooperative false positive suppression feature requires the sharing of canoncalized configuration entries, which is easier to perform in a centralized-management setting.

Future work: Our experiments showed that the noise in event logs varied greatly from program to program—after all, these programs have different purposes, workloads, and users. Currently CODE treats all programs uniformly in learning. However, we envision harnessing program-specific knowledge to further improve our detection accuracy and reduce false positives. In particular we may set a higher support threshold for a noisier program. Another possibility is to rank errors based on the importance of the programs affected by these errors. For example, a warning from `system.exe` (the Windows kernel process) may be more important than a warning from `explore.exe`.

In a distributed setting, CODE can collect a much larger, unbiased set of logs to improve the quality of its rules. In particular, for managing server clusters, the homogeneity of the machines may also help reduce CODE's memory overhead and false positive rates (see Section 6, 7.2, and 7.3). One challenge is canonicalization: the rules CODE learns may contain machine-specific information (e.g. machine names, IP addresses, and user names). We manually added user-name canonicalization in CODE. As future work, we plan to develop automatic or semi-automatic techniques to infer more machine-specific configuration data for canonicalization.

9 Conclusion

We presented CODE, an online, automatic tool for configuration error detection. Our observation is rather simple: key configuration access events form highly repetitive sequences. These sequences are much more deterministic than each individual event, thus can serve as contexts to predict future events. Based on this observation, CODE uses a context-based analysis to efficiently analyze a massive amount of configuration events. We implemented CODE on Windows and used it to detect Windows Registry errors. Our results showed that CODE could successfully detect real-world configuration errors with a low false positive rate and low runtime overhead.

Acknowledgments

We thank the anonymous reviewers and our paper shepherd Dilma Da Silva for their valuable feedbacks. We also thank Marcos K. Auguilera for his detailed comments for improving the paper. We thank Professor Yuanyuan Zhou, the UCSD Opera research group and Marti Motoyama for discussion and paper proofreading.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering (ICDE)*, pages 3–14, 1995.

[2] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. of ACM*, 13(7):422–426, 1970.

[4] G. Candea. Toward quantifying system manageability. In *Proceedings of the Fourth conference on Hot topics in system dependability (HotDep)*, 2008.

[5] Description of Office 2003 service pack 3. `http://support.microsoft.com/kb/923618`.

[6] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang. Automatic software fault diagnosis by exploiting application signatures. In *Proceedings of the 22nd conference on Large installation system administration conference (LISA)*, pages 23–39, 2008.

[7] A. Ganapathi, Y.-M. Wang, N. Lao, and J.-R. Wen. Why PCs are fragile and what we can do about it: A study of Windows registry problems. In *DSN'04*.

[8] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 101–111, 2007.

[9] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[10] Windows still No.1 in server OS. http://www.zdnet.com/blog/microsoft/behind-the-idc-data-windows-still-no-1-in-server-operating-systems/5408.

[11] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *Proceedings of IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 157–166, 2008.

[12] E. Kiciman and Y.-M. Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC)*, 2004.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[14] N. Kushman and D. Katabi. Enabling configuration-independent automation by non-expert users. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.

[15] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. *SIGSOFT Softw. Eng. Notes*, 30:306–315, September 2005.

[16] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2), 1976.

[17] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 1997.

[18] M. O. Rabin. Fingerprinting by random polynomials. In *Harvard University Report TR-15-81 (1981)*.

[19] J. A. Redstone, M. M. Swift, and B. N. Bershad. Using computers to diagnose computer problems. In *Proceedings of the 9th conference on Hot Topics in Operating Systems (HotOS)*, 2003.

[20] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 237–250, 2007.

[21] Symantec hotbar adware information. `http://www.symantec.com/security_response/writeup.jsp?docid=2003-080410-3847-99`.

[22] Top 5 OSes on Oct 09. `http://gs.statcounter.com/#os-ww-monthly-200910-200910-bar`.

[23] C. Verbowski, E. Kiciman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight data recorder: Monitoring persistent-state interactions to improve systems management. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[24] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[25] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and communications security (CCS)*, pages 255–264, 2002.

[26] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[27] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, and C. Yuan. Strider: A black-box, state-based approach to change and configuration management and support. In *Proceedings of the 17th USENIX conference on System administration*, pages 159–172, 2003.

[28] L. R. Welch. Hidden markov models and the Baum-Welch algorithm. *IEEE Info. Theory Society Newsletter*, 2003.

[29] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[30] Windows automatic update disabled. `http://forums.lenovo.com/t5/Windows-XP-and-Vista-discussion/Windows-automatic-update-disabled/td-p/69380`.

[31] Operating system market share. `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8`.

[32] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, pages 375–388, 2006.

[33] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, pages 143–154.

OFRewind: Enabling Record and Replay Troubleshooting for Networks

Andreas Wundsam[*] *Dan Levin*[*] *Srini Seetharaman*[‡] *Anja Feldmann*[*]

[*] *Deutsche Telekom Laboratories / TU Berlin, {andi,dan,anja}@net.t-labs.tu-berlin.de*

[‡] *Deutsche Telekom Inc., R&D Lab USA, srini.seetharaman@telekom.com*

Abstract

Debugging operational networks can be a daunting task, due to their size, distributed state, and the presence of *black box* components such as commercial routers and switches, which are poorly instrumentable and only coarsely configurable. The debugging tool set available to administrators is limited, and provides only aggregated statistics (SNMP), sampled data (NetFlow/sFlow), or local measurements on single hosts (tcpdump). In this paper, we leverage *split forwarding architectures* such as OpenFlow to add *record and replay debugging* capabilities to networks – a powerful, yet currently lacking approach. We present the design of **OFRewind**, which enables *scalable*, *multi-granularity*, *temporally consistent recording* and *coordinated replay* in a network, with fine-grained, dynamic, centrally orchestrated control over both record and replay. Thus, **OFRewind** helps operators to reproduce software errors, identify data-path limitations, or locate configuration errors.

1 Introduction

Life as a network operator can be hard. In spite of many efforts to the contrary, problem localization and troubleshooting in operational networks still remain largely unsolved problems today. Consider the following anecdotal evidence:

Towards the end of October 2009, the administrators of the Stanford production OpenFlow network began observing strange CPU usage patterns in their switches. The CPU utilization oscillated between 25% and 100% roughly every 30 minutes and led to prolonged flow setup times, which where unacceptable for many users. The network operators began debugging the problem using standard tools and data sets, including SNMP statistics, however the cause for the oscillation of the switch CPU remained inexplicable. Even an analysis of the entire control channel data could not shed light on the cause of the problem, as no observed parameter (number of: packets in, packets out, flow modifications, flow expirations, statistics requests, and statistics replies) seemed to correlate with the CPU utilization. This left the network operator puzzled regarding the cause of the problem.

This anecdote (further discussion in Section 4.2) hints at some of the challenges encountered when debugging problems in networks. Networks typically contain *black box* devices, e.g., commercial routers, switches, and middleboxes, that can be only coarsely configured and instrumented, via command-line or simple protocols such as SNMP. Often, the behavior of black box components in the network cannot be understood by analytical means alone – controlled replay and experimentation is needed.

Furthermore, network operators remain stuck with a fairly simplistic arsenal of tools. Many operators record statistics via NetFlow or sFlow [33]. These tools are valuable for observing general traffic trends, but often too coarse to pinpoint the origin problems. Collecting full packet traces, e.g., by tcpdump or specialized hardware, is often unscalable due to high volume data plane traffic. Even when there is a packet trace available, it typically only contains the traffic of a single VLAN or switch port. It is thus difficult to infer temporal or causal relationships between messages exchanged between multiple ports or devices.

Previous attempts have not significantly improved the situation. Tracing frameworks such as XTrace [35] and Netreplay [11] enhance debugging capabilities by pervasively instrumenting the entire network ecosystem, but face serious deployment hurdles due to the scale of changes involved. There are powerful tools available in the context of distributed applications that enable fully deterministic recording and replay, oriented toward end hosts [16, 24]. However, overhead for the fully-deterministic recording of a large network with high data rates can be prohibitive and the instrumentation of 'black' middleboxes and closed source software often remains out of reach.

In this paper, we present a new approach to enable practical network recording and replay, based upon an emerging class of network architectures called *split forwarding architectures*, such as OpenFlow [28], Tesseract [41], and Forces [2]. These architectures *split* control plane decision-making off from data plane forwarding. In doing so, they enable custom programmability and centralization of the control plane, while allowing for commodity high-throughput, high-fanout data plane forwarding elements.

We discuss, in Section 2, the design of **OFRewind**, a tool that takes advantage of these properties to significantly improve the state-of-the-art for recording and replaying network domains. **OFRewind** enables *scalable*, *temporally consistent*, *centrally controlled* network recording and *coordinated* replay of traffic in an OpenFlow controller domain. It takes advantage of the flexibility afforded by the programmable control plane, to dynamically *select* data plane traffic for recording. This improves data-path component scalability and enables *always-on* recording of critical, low-volume traffic, e.g., routing control messages. Indeed, a recent study has shown that the control plane traffic accounts for less than 1% of the data volume, but $95 - 99\%$ of the observed bugs [10]. Data plane traffic can be *load-balanced* across multiple *data plane recorders*. This enables recording even in environments with high data rates. Finally, thanks to the centralized perspective of the controller, **OFRewind** can record a *temporally consistent* trace of the controller domain. This facilitates investigation of the temporal and causal interdependencies of the messages exchanged between the devices in the controller domain.

During replay, **OFRewind** enables the operator to select which parts of the traces are to be replayed and how they should be mapped to the replay environment. By partitioning (or *bisecting*) the trace and automatically repeating the experiment, our tool helps to narrow down and isolate the problem causing component or traffic. A concrete implementation of the tool based on OpenFlow is presented in Section 3 and is released as free and open source software [4].

Our work is primarily motivated by operational issues in the OpenFlow-enabled production network at Stanford University. Accordingly, we discuss several case studies where our system has proven useful, including: switch CPU inflation, broadcast storms, anomalous forwarding, NOX packet parsing errors, and other invalid controller actions (Section 4). We in addition present a case study in which **OFRewind** successfully pinpoints faulty behavior in the Quagga RIP software routing daemon. This indicates that **OFRewind** is not limited to locating OpenFlow-specific bugs alone, but can also be used to reproduce other network anomalies.

Our evaluation (Section 5) shows *(a)* that the tool scales at least as well as current OpenFlow hardware implementations, *(b)* that recording does not impose an undue performance penalty on the throughput achieved, and *(c)* that the messaging overhead for synchronization in our production network is limited to 1.13% of all data plane traffic.

While using our tool, we have made and incorporated the following key observations:

(1) A full recording of all events in an entire production network is infeasible, due to the data volumes involved and their asynchronous nature. However, one usually needs not record all information to be able to reproduce or pinpoint a failure. It suffices to focus on *relevant subparts*, e.g., control messages or packet headers. By selectively recording critical traffic subsets, we can afford to turn *recording on by default* and thus reproduce many unforeseen problems *post facto*.

(2) Partial recordings, while missing some data necessary for fully deterministic replay, can be used to reproduce symptomatic network behavior, useful for gaining insights in many debugging situations. With careful initialization, the behavior of many network devices turns out to be *deterministic with respect to the network input*.

(3) By replaying *subsets of traffic* at a *controlled pace*, we can, in many cases, rapidly repeat experiments with different settings (parameters/code hooks) while still reproducing the error. We can, for instance, *bisect* the traffic and thus localize the sequence of messages leading to an error.

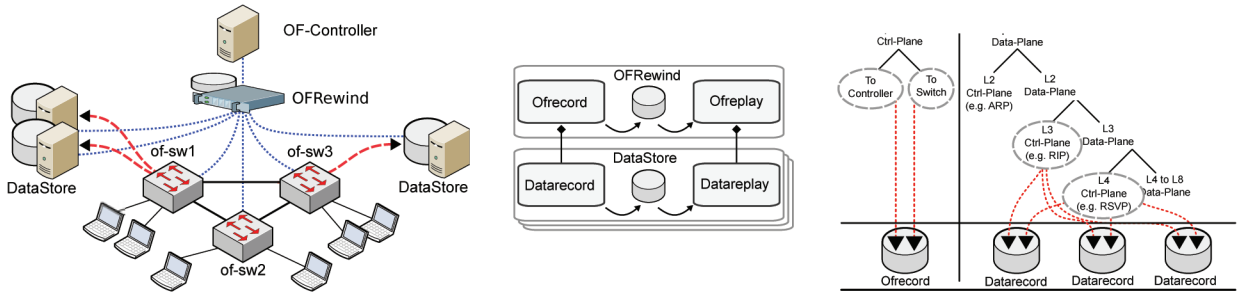
In summary, **OFRewind** is, to the best of our knowledge, the first tool which leverages the properties of split architecture forwarding to enable practical and economically feasible recording and replay debugging of network domains. It has proven useful in a variety of practical case studies, and our evaluation shows **OFRewind** does not significantly affect the scalability of OpenFlow controller domains and does not introduce undue overhead.

2 OFRewind System Design

In this section, we discuss the expected operational environment of **OFRewind**, its design goals, and the components and their interaction. We then focus on the need to synchronize specific system components during operation.

2.1 Environment / Abstractions

We base our system design upon *split forwarding architectures*, for instance, OpenFlow [28], Tesseract [41], or Forces [2], in which *standardized data plane elements* (switches) perform fast and efficient packet forwarding, and the control plane is *programmable* via an external



(a) System components (b) Software modules (c) Traffic strata

Figure 1: Overview of **OFRewind**

controlling entity, known as the *controller*. Programmability is achieved through *forwarding rules* that *match* incoming traffic and associate them with *actions*. We call this layer of the network the *substrate*, and the higher-layer *superstrate* network running on top of it *guest*. We call the traffic exchanged between the switches and the controller the substrate *control plane*. The higher-layer control plane traffic running inside of the substrate *data plane* (e.g., IP routing messages) is called the guest *control plane*. The relationship between these layers and traffic strata is shown in Figure 1(c).

Even though not strictly mandated almost all split-architecture deployments group several switches to be centrally managed by one controller, creating a *controller domain*. We envision one instance of **OFRewind** to run in one such controller domain. Imagine, e.g., a campus building infrastructure with 5-10 switches, 200 hosts attached on Gigabit links, a few routers, and an uplink of 10GB/s.

2.2 Design Goals and Non-Goals

As previously stated, recording and replay functionalities are not usually available in networks. We aim to build a tool that leverages the flexibility afforded by split-architectures to realize such a system. We do not envision **OFRewind** to do automated *root-cause analysis*. We do intend it to help *localize* problem causes. Additionally, we do not envision it to *automatically tune recording* parameters. This is left to an informed administrator who knows what scenario is being debugged.

Recording goals: We want a *scalable* system that can be used in a realistic-sized controller domain. We want to be able to record critical traffic, e.g., routing messages, in an *always-on* fashion. What is monitored should be specified in *centralized configuration*, and we want to be able to attain a *temporally consistent* view of the recorded events in the controller domain.

Replay goals: We want to be able to replay traffic in a *coordinated* fashion across the controller domain. For replaying into a different environment or topology (e.g., in a lab environment) we want to *sub-select* traffic and potentially *map* traffic to other devices. We include *time dilation* to help investigate timer issues, create stress tests, and allow "fast forwarding" to skip over irrelevant portions of the traffic. *Bisection* of the traffic between replays can assist problem localization whereby the user repeatedly partitions and sub-selects traffic to be replayed based on user-defined criteria (e.g., by message types), performs a test run, then continues the bisection based on whether a problem was reproducible.

(Absence of) determinism guarantees: As opposed to host-oriented replay debugging systems which strive for determinism guarantees, **OFRewind** does not – and cannot – provide strict determinism guarantees, as *black boxes* do not lend themselves to the necessary instrumentation. Instead, we leverage the insight that network device behavior is *largely deterministic* on control plane events (messages, ordering, timing). In some cases, when devices deliberately behave non-deterministically, protocol specific approaches must be taken.

2.3 OFRewind System Components

As seen in Figure 1(a), the main component of our system, **OFRewind**, runs as a proxy on the substrate control channel, i.e., between the switches and the original controller. It can thus intercept and modify substrate control plane messages to control recording and replay. It delegates recording and replay of *guest* traffic to *DataStore* components that are locally attached at regular switch ports. The number of *DataStores* attached at each switch can be chosen freely, subject to the availability of ports.

Both components can be broken down further into two modules each, as depicted in Figure 1(b): They consist of a recording and a replay module with a shared local storage, labeled *Ofrecord* and *Ofreplay*, and *Datarecord* and *Datareplay* respectively.

Record: *Ofrecord* captures *substrate* control plane traffic directly. When *guest* network traffic recording is

desired, *Ofrecord* translates control messages to instruct the relevant switches to selectively mirror *guest* traffic to the *Datarecord* modules. **OFRewind** supports dynamic selection of the *substrate* or *guest* network traffic to record. In addition, flow-based-sampling can be used to record only a fraction of the data plane flows.

Replay: *Ofreplay* re-injects the traces captured by *Ofrecord* into the network, and thus enables domain-wide *replay debugging*. It emulates a controller towards switches or a set of switches towards a controller, and directly replays *substrate* control plane traffic. *Guest* traffic is replayed by the *Datareplay* modules, which are orchestrated by *Ofreplay*.

2.4 *Ofrecord* Traffic Selection

While it is, in principle, possible to collect full data recordings for every packet in the network, this introduces a substantial overhead both in terms of storage as well as in terms of performance. *Ofrecord*, however, allows selective traffic recording to reduce the overhead.

Selection: Flows can be classified and selected for recording. We refer to traffic *selection* whenever we make a conscious decision on what subset of traffic to record. Possible categories include: *substrate* control traffic, *guest* network control traffic, or *guest* data traffic, possibly sub-selected by arbitrary match expressions. We illustrate an example selection from these categories in Figure 1(c).

Sampling: If selection is unable to reduce the traffic sufficiently, one may apply either packet or flow sampling on either type of traffic as a reduction strategy.

Cut-offs: Another data reduction approach is to record only the first X bytes of each packet or flow. This often suffices to capture the critical meta-data and has been used in the context of intrusion detection [26].

2.5 *Ofreplay* Operation Modes

To support testing of the different entities involved (switches, controller, end hosts) and to enable different playback scenarios, *Ofreplay* supports several different operation modes:

ctrl: In this operation mode, replay is directed towards the controller. *Ofreplay* plays the recorded substrate control messages from the local *storage*. This mode enables debugging of a controller application on a single developer host, without need for switches, end-hosts, or even a network. Recorded data plane traffic is not required.

switch: This operation mode replays the recorded *substrate* control messages toward the switches, reconstructing each switch's *flow table* in real time. No controller is needed, nor is *guest* traffic replayed.

datahdr: This mode uses packet headers captured by the *Datarecord* modules to re-generate the exact flows encountered at recording time, with dummy packet payloads. This enables full testing of the switch network, independent of any end hosts.

datafull: In this mode, data traffic recorded by the *DataStores* is replayed with complete payload, allowing for selective inclusion of end host traffic into the tests.

In addition to these operation modes, *Ofreplay* enables the user to further tweak the recorded traffic to match the replay scenario. Replayed messages can be *sub-selected* based on source or destination host, port, or message type. Further, message destinations can be *re-mapped* on a per-host or per-port basis. These two complementary features allow traffic to be re-targeted toward a particular host, or restricted such that only relevant messages are replayed. They enable *Ofreplay* to play recorded traffic either toward the original sources or to alternative devices, which may run a different firmware version, have a different hardware configuration, or even be of a different vendor. These features enable **OFRewind** to be used for *regression testing*. Alternately, it can be useful to map traffic of multiple devices to a single device, to perform stress tests.

The *pace* of the replay is also adjustable within *Ofreplay*, enabling investigation of pace-dependent performance problems. Adjusting replay can also be used to "fast-forward" over portions of a trace, e.g., memory leaks in a switch, which typically develop over long time periods may be reproduced in an expedited manner.

2.6 Event Ordering and Synchronization

For some debugging scenarios, it is necessary to preserve the exact message order or mapping between *guest* and *substrate* flow data to be able to reproduce the problem. In concrete terms, the guest (data) traffic should not be replayed until the substrate (control) traffic (containing the corresponding substrate actions) has been replayed. Otherwise, *guest* messages might be incorrectly forwarded or simply dropped by the switch, as the necessary flow table state would be invalid or missing.

We do not assume tightly synchronized clocks or low latency communication channels between our **OFRewind** and the *DataStores* components. Accordingly, we cannot assume that synchronization between recorded *substrate* and *guest* flow traces, or order between flows recorded by different *DataStores* is guaranteed per se. Our design does rely on the following assumptions: (1) The *substrate* control plane channel is reliable and order-preserving. (2) The *control channel* between **OFRewind** and each individual *DataStore* is reliable and order-preserving, and has a reasonable mean latency l_c (e.g., 5 ms in a LAN setup.) (3) The *data plane* channel from **OFRewind** to the *DataStores* via the switch is not necessarily fully, but sufficiently reliable (e.g., 99.9% of mes-

sages arrive). It is not required to be order-preserving in general, but there should be some means of explicitly guaranteeing order between two messages. We define the data plane channel mean latency as l_d.

Record: Based on these assumptions, we define a logical clock C [25] on *Ofrecord*, incremented for each *substrate* control message as they arrive at *Ofrecord*. *Ofrecord* logs the value of C with each *substrate* control message. It also broadcasts the value of C to the *DataStores* in two kinds of synchronization markers: *time binning markers* and *flow creation markers*.

Time binning markers are sent out at regular time intervals i_t, e.g., every 100ms. They group flows into bins and thus constrain the search space for matching flows during replay and help reconstruct traffic characteristics within flows. Note that they do not impose a strict order on the flows within a time bin.

Flow creation markers are optionally sent out whenever a new flow is created. Based on the previous assumptions, they induce a total ordering on all flows whose creation markers have been successfully recorded. However, their usage limits the scalability of the system, as they must be recorded by all *DataStores*.

Replay: For synchronization during replay, *Ofreplay* assumes the role of a synchronization master, reading the value of C logged with the *substrate* messages. When a *DataStore* hits a synchronization marker while replaying, it synchronizes with *Ofreplay* before continuing. This assures that in the presence of *time binning markers*, the replay stays loosely synchronized between the markers (within an interval $I = i_t + l_d + l_c$). In the presence of *flow creation markers*, it guarantees that the order between the marked flows will be preserved.

2.7 Typical Operation

We envision that users of **OFRewind** run *Ofrecord* in an always-on fashion, always recording selected *substrate* control plane traffic (e.g., OpenFlow messages) onto a ring storage. If necessary, selected *guest* traffic can also be continuously recorded on *Datarecord*. To preserve space, low-rate control plane traffic, e.g., routing announcements, may be selected, sampling may be used, and/or the ring storage may be shrunk. When the operator (or an automated analytics tool) detects an anomaly, a replay is launched onto a separate set of hardware, or onto the production network during off-peak times. Recording settings are adapted as necessary until the anomaly can be reproduced during replay.

During replay, one typically uses some kind of debugging by elimination, either by performing binary search along the time axis or by eliminating one kind of message at a time. Hereby, it is important to choose orthogonal subsets of messages for effective problem localization.

3 Implementation

In this section, we describe the implementation of **OFRewind** based on OpenFlow, selected for currently being the most widely used *split forwarding architecture*. OpenFlow is currently in rapid adoption by testbeds [20], university campuses [1], and commercial vendors [3].

OpenFlow realizes *split forwarding architecture* as an open protocol between packet-forwarding hardware and a commodity PC (the *controller*). The protocol allows the *controller* to exercise flexible and dynamic control over the forwarding behavior of OpenFlow enabled Ethernet switches at a per-flow level. The definition of a flow can be tailored to the specific application case—OpenFlow supports an 11-tuple of packet header parts, against which incoming packets can be matched, and flows classified. These range from Layer 1 (switch ports), to Layer 2 and 3 (MAC and IP addresses), to Layer 4 (TCP and UDP ports). The set of matching rules, and the actions associated with and performed on each match are held in the switch and known as the *flow table*.

We next discuss the implementation of **OFRewind**, the synchronization among the components and discuss the benefits, limitations, and best-practices of using OpenFlow to implement our system. The implementation, which is an OpenFlow controller in itself, and based on the source code of FlowVisor [36] is available under a free and open source license at [4].

3.1 Software Modules

To capture both the *substrate* control traffic and *guest* network traffic we use a hybrid strategy for implementing **OFRewind**. Reconsider the example shown in Figure 1(a) from an OpenFlow perspective. We deploy a proxy server in the OpenFlow protocol path (labeled **OFRewind**) and attach local *DataStore* nodes to the switches. The **OFRewind** node runs the *Ofrecord* and *Ofreplay* modules, and the *DataStore* nodes run *Datarecord* and *Datareplay*, respectively. We now discuss the implementation of the four software components *Ofrecord*, *Datarecord*, *Ofreplay* and *Datareplay*.

***Ofrecord*:** *Ofrecord* intercepts all messages passing between the switches and controller and applies the selection rules. It then stores the selected OpenFlow control (*substrate*) messages to locally attached data storage. Optionally, the entire flow table of the switch can be dumped on record startup. If recording of the *guest* network control and/or data traffic is performed, *Ofrecord* transforms the `FLOW-MOD` and `PACKET-OUT` commands sent from the controller to the switch to *duplicate* the packets of selected flows to a *DataStore* attached to a switch along flow path. Multiple *DataStores* can be attached to each switch, .e.g., for load-balancing. The order of flows on the different *DataStores* in the sys-

tem is retained with the help of synchronization markers. Any match rule supported by OpenFlow can be used for packet selection. Additionally, flow-based-sampling can be used to only record a fraction of the flows.

***Datarecord*:** The *Datarecord* components located on the *DataStores* record the selected guest traffic, as well as synchronization and control metadata. They are spawned and controlled by *Ofrecord*. Their implementation is based on `tcpdump`, modified to be controlled by *Ofrecord* via a TCP socket connection. Data reduction strategies that cannot be implemented with OpenFlow rules (e.g., packet sampling, cut-offs) are executed by *Datarecord* before writing the data to disk.

***Ofreplay*:** *Ofreplay* re-injects OpenFlow control plane messages as recorded by *Ofrecord* into the network and orchestrates the guest traffic replay by the *Datareplay* components on the *DataStores*. It supports replay towards the *controller* and *switches*, and different levels of data plane involvement (*switch*, *datahdr*, *datafull*, see Section 2.5.) Optionally, a flow table dump created by *Ofrecord* can be installed into the switches prior to replay. It supports traffic *sub-selection* and *mapping* towards different hardware and *time dilation*.

***Datareplay*:** The *Datareplay* components are responsible for re-injecting guest traffic into the network. They interact with and are controlled by *Ofreplay* for timing and synchronization. The implementation is based on `tcpreplay`. Depending on the record and replay mode, they reconstruct or synthesize missing data before replay, e.g., dummy packet payloads, when only packet headers have been recorded.

3.2 Synchronization

As we do not assume precise time synchronization between *Ofrecord* and the *DataStores*, the implementation uses *time binning markers* and *flow creation markers*, as discussed in Section 2.6. These are packets with unique ids flooded to all *DataStores* and logged by *Ofrecord*. The ordering of these markers relative to the recorded traffic is ensured by OpenFlow `BARRIER` messages[1]. We now discuss by example how the markers are used.

Record synchronization: Figure 2(a) illustrates the use of *flow creation markers* during recording. Consider a simple *Ofrecord* setup with two hosts *c1* and *s1* connected to switch *sw*. The switch is controlled by an instance of *Ofrecord*, which in turn acts as a client to the network controller *ctrl*. *Ofrecord* records to the local storage *of-store*, and controls an instance of *Datarecord* running on a *DataStore*. Assume that a new TCP connection is initiated at *c1* toward *s1*, generating a *tcp syn* packet (Step 1). As no matching flow table entry exists, *sw* sends *msg1*, an OpenFlow `PACKET-IN` to *Ofrecord* (Step 2), which in turn relays it to *ctrl* (step 3). *Ctrl* may respond with *msg2*, a `FLOW-MOD` message (step 4). To enable synchronized replay and reassembly of the control and data records, *Ofrecord* now creates a flow creation marker (*sync*), containing a unique id, the current time, and the matching rule of *msg1* and *msg2*. Both *msg1* and *msg2* are then annotated with the id of *sync* and saved to *of-store* (step 5). *Ofrecord* then sends out 3 messages to *sw1*: first, a `PACKET-OUT` message containing the *flow creation marker* sent to *all DataStores* in step 6. This prompts the switch to send out *sync* to all its attached *DataStores* (step 7). The second message sent from *Ofrecord* is a `BARRIER` message (step 8), which ensures that the message from step 7 is handled before any subsequent messages. In step 9, *Ofrecord* sends a modified `FLOW-MOD` message directing the flow to both the original receiver, as well as *one DataStore* attached to the switch. This prompts the switch to output the flow both to *s1* (step 10a) and *DataStore* (step 10b).

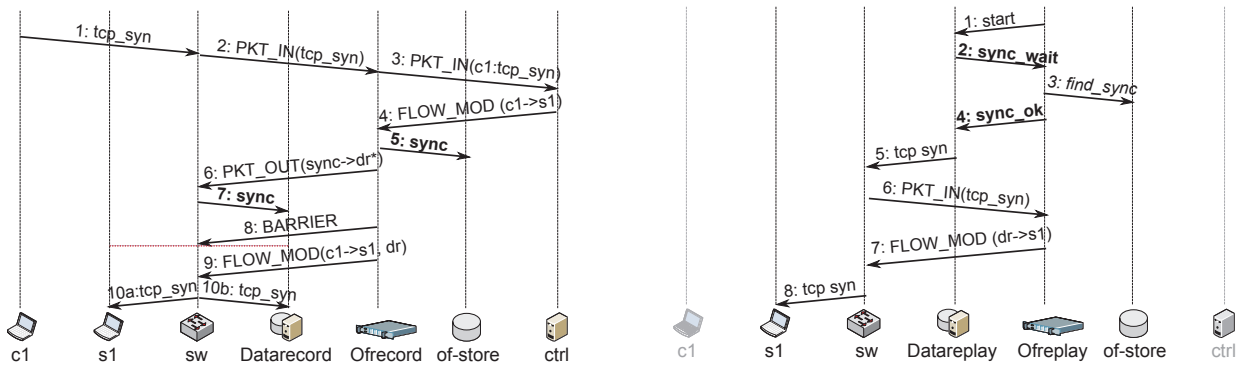
Replay synchronization: For synchronizing replay, *Ofreplay* matches data plane events to control plane events with the help of the flow creation markers recorded by *Ofrecord*. Consider the example in Figure 2(b). Based on the previous example, we replay the recording in *data plane mode* towards the switch *sw* and host *s1*. To begin, *Ofreplay* starts *Datareplay* playback on the *DataStore* in step 1. *Datareplay* hits the flow creation marker *sync*, then sends a `sync_wait` message to the controller, and goes to sleep (step 2). *Ofrecord* continues replay operation, until it hits the corresponding flow creation marker *sync* on the *of-store* (step 3). Then, it signals *Datareplay* to continue with a `sync_ok` message (step 4). *Datareplay* outputs the packet to the switch (step 5), generating a `PACKET-IN` event (step 6). *Ofreplay* responds with the matching `FLOW-MOD` event from the OpenFlow log (step 7). This installs a matching flow rule in the switch and causes the flow to be forwarded as recorded (step 8).

3.3 Discussion

We now discuss the limitations imposed by OpenFlow on our implementation, and best practices for avoiding replay inaccuracies.

OpenFlow limitations: While OpenFlow provides a useful basis for implementing **OFRewind**, it does not support all the features required to realize all operation modes. OpenFlow does not currently support **sampling** of either flows or packets. Thus, *flow sampling* is performed by *Ofrecord*, and packet sampling is performed by *Datarecord*. This imposes additional load on the channel between the switches and the *DataStores*

[1] A BARRIER message ensures that all prior OpenFlow messages are processed before subsequent messages are handled. In its absence, messages may be reordered.



(a) *Ofrecord* (b) *Ofreplay*

Figure 2: *DataStore* synchronization mechanism in **OFRewind**

for data that is not subsequently recorded. Similarly, the OpenFlow data plane does not support forwarding of **partial packets**[2]. Consequently, full packets are forwarded to the *DataStore* and only their headers may be recorded. OpenFlow also does not support automatic **flow cut-offs** after a specified amount of *traffic*[3]. The cut-off can be performed in the *DataStore*. Further optimizations are possible, e.g., regularly removing flows that have surpassed the threshold.

Avoiding replay inaccuracies: To reliably reproduce failures during replay in a controlled environment, one must ensure that the environment is properly initialized. We suggest therefore, to use the flow table dump feature and, preferably, reset (whenever possible) the switches and controller state before starting the replay operation. This reduces any unforeseen interference from previously installed bad state.

When bisecting during replay, one must consider the interdependencies among message types. `FLOW-MOD` messages are for example, responsible for creating the flow table entries and their arbitrary bisection may lead to incomplete or nonsense forwarding state on the switch.

Generally speaking, replay inaccuracies can occur when: (a) the chain of causally correlated messages is recorded incompletely, (b) synchronization between causally correlated messages is insufficient, (c) timers influence system behavior, and (d) network communication is partially non-deterministic. For (a) and (b), it is necessary to adapt the recording settings to include more or better-synchronized data. For (c) a possible approach is to *reduce* the traffic being replayed via subselection to reduce stress on the devices and increase accuracy. We have not witnessed this problem in our practical case studies. Case (d) requires the replayed traffic to be modified. If the non-determinism stems from the transport layer (e.g., TCP random initial sequence numbers), a custom transport-layer handler in *Datareplay* can shift sequence numbers accordingly for replay. For application non-determinism (e.g., cryptographic nonces), application-specific handlers must be used.

When the failure observed in the production network does not appear during replay, we call this a **false negative** problem. When the precautions outlined above have been taken, a repeated *false negative* indicates that the failure is likely not triggered by network traffic, but other events. In a **false positive** case, a failure is observed during replay which does not stem from the same root cause. Such inaccuracies can often be avoided by careful comparison of the symptoms and automated repetition of the replay.

4 Case Studies

In this section, we demonstrate the use of **OFRewind** for localizing problems in *black box network devices*, *controllers*, and *other software components*, as summarized in Table 1. These case studies also demonstrate the benefits of *bisecting* the control plane traffic (4.2), of *mapping* replays onto different pieces of hardware (4.3), from a production network onto a developer machine (4.5), and the benefit of a *temporally consistent* recording of multiple switches (4.6).

4.1 Experimental Setup

For our case studies we use a network with switches from three vendors: **Vendor A**, **Vendor B**, **Vendor C**. Each switch has two PCs connected to it. Figure 3 illustrates the connectivity. All switches in the network have a control-channel to *Ofrecord*. *DataStores* running *Datarecord* and *Datareplay* are attached to the switches as necessary. We use NOX [31], unless specified otherwise, as the high level controller performing the actual routing decisions. It includes the $routing$ module, which

[2]It *does* support a cut-off for packets forwarded to the *controller*.
[3]Expiration after a specified amount of *time* is supported.

Case study	Class	OF-specific
Switch CPU Inflation	black box (switch)	no
Anomalous Forwarding	black box (switch)	yes
Invalid Port Translation	OF controller	yes
NOX Parsing Error	OF controller	yes
Faulty Route Advertisements	software router	no

Table 1: Overview of the case studies

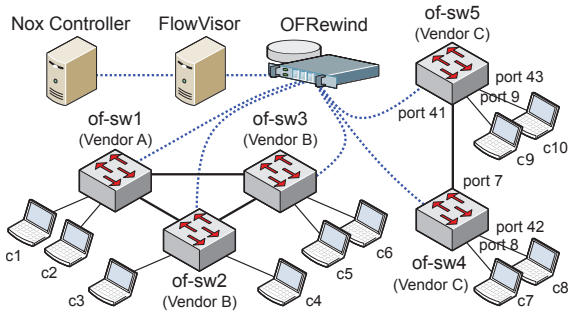


Figure 3: Lab environment for case studies

performs shortest path routing by learning the destination MAC address, and the $spanning - tree$ module, which prevents broadcast storms in the network by using Link Layer Discovery Protocol (LLDP) to identify if there is a loop in the network. All OpenFlow applications, viz. NOX, FlowVisor, *Ofreplay*, and *Ofrecord*, are run on the same server.

4.2 Switch CPU Inflation

Figure 4 shows our attempt at reproducing the CPU oscillation we reported in Section 1. As stated earlier, there is no apparent correlation between the ingress traffic and the CPU utilization. We record all control traffic in the production network, as well as the traffic entering/exiting the switch, while the CPU oscillation is happening. Figure 4(a) shows the original switch performance during recording. We, then, iteratively replay the corresponding control traffic over a similar switch in our isolated experimental setup. We replay the recorded data traffic to 1 port of the switch and connect hosts that send `ICMP` datagrams to the other ports. In each iteration, we have *Ofreplay bisect* the trace by OpenFlow message type, and check whether the symptom persists. When replaying the port and table statistic requests, we observe the behavior as shown in Figure 4(b). Since the data traffic is synthetically generated, the amplitude of the CPU oscillation and the flow setup time variation is different from that in the original system. Still, the sawtooth pattern is clearly visible. This successful reproduction of the symptom helps us identify the issue to be related to port and table statistics requests. Note that these messages have been causing the anomaly, even though their

Counts	Match
duration=181s	in_port=8
n_packets=0	dl_type=arp
n_bytes=3968	dl_src=00:15:17:d1:fa:92
idle_timeout=60	dl_dst=ff:ff:ff:ff:ff:ff
hard_timeout=0	actions=`FLOOD`

Table 2: **Vendor C** switch flow table entry, during replay.

arrival rate (24 messages per minute) is not in any way temporally correlated with the perceived symptoms (30-minute CPU sawtooth pattern). We reported this issue to the vendor, since at this point we have no more visibility into the switch software implementation.

OFRewind thus, has proved useful in localizing the cause for the anomalous behavior of a black box component that would otherwise have been difficult to debug. Even though the bug in this case is related to a prototype OpenFlow device, the scenario as such (misbehaving black box component) and approach (debugging by replay and bisection of control-channel traffic) are arguably applicable to non-OpenFlow cases as well.

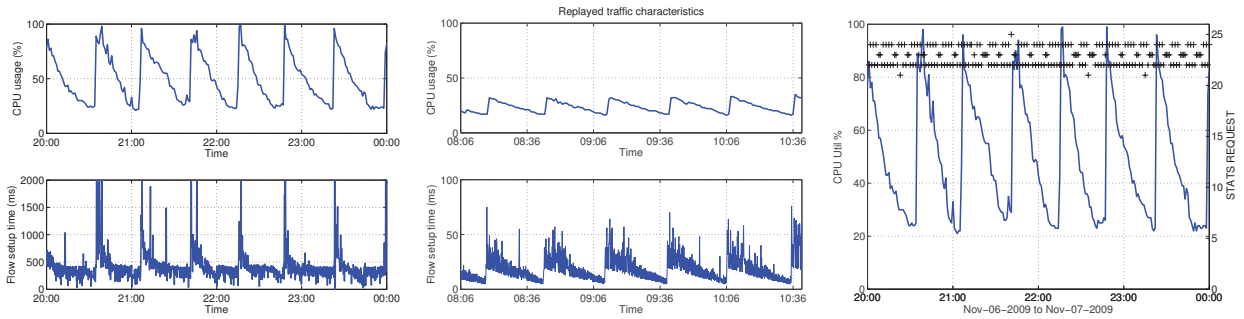
4.3 Anomalous Forwarding

To investigate the performance of devices from a new vendor, **Vendor C**, we record the substrate and guest traffic for a set of flows, sending 10 second delayed `ping` between a pair of hosts attached to the switch from **Vendor B** (`of-sw3` in Figure 3). We then use the device/port mapping feature of *Ofreplay* to play back traffic from $c7$ to $c8$ over port 8 and port 42 belonging to the switch from **Vendor C**, in Figure 3.

Upon replay, we observe an interesting limitation of the switch from **Vendor C**. The `ping` flow stalls at the ARP resolution phase. The ARP packets transmitted from host $c7$ are received by host $c10$, but not by $c8$ nor $c9$. The flow table entry created in `of-sw4` during replay, is shown in Table 2, similar to that created during the original run. We conclude that the `FLOOD` action is not being properly applied by the switch from **Vendor C**.

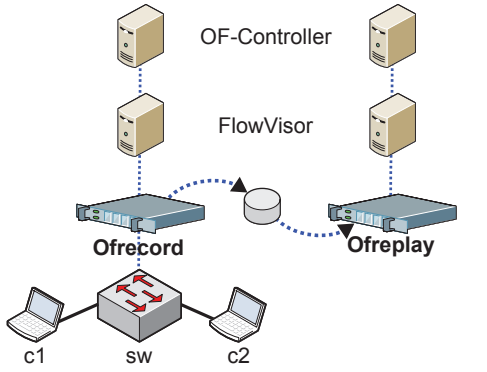
Careful observation reveals that traffic received on a "low port" (one of the first 24 ports) to be flooded to any "high ports" (last 24 ports) and vice-versa is not flooded correctly. Effectively, the flood is restricted within a 24 port group within the switch (lower or higher). This fact has been affirmed by the vendor, confirming the successful debugging.

We additionally perform the replay after adding static ARP entries to the host $c7$. In this case, we observe that flow setup time for the subsequent unicast `ping` traffic on **Vendor C** is consistently higher than that observed for **Vendor B** and **Vendor A** switches. This indicates that **OFRewind** has further potential in profiling switches and controllers.



(a) Switch performance with original traffic (b) Switch performance with replayed traffic (c) `STATS-REQUEST` vs. CPU

Figure 4: Sawtooth CPU pattern reproduced during replay of port and table `STATS-REQUEST` messages. Figure (c) shows no observable temporal correlation to message arrivals.



(a) recording the bug (b) replay of bug

Figure 5: Debugging of FlowVisor bug #68

4.4 Invalid Port Translation

In this case study, we operate *Ofreplay* in the *ctrl* mode in order to debug a controller issue. The controller we focus on is the publicly available FlowVisor [36].

FlowVisor (FV) is a special purpose OpenFlow controller that acts as a proxy between multiple OpenFlow switches and controllers (*guests*), and thus assumes the role of a hypervisor for the OpenFlow control plane (see Figure 3). To this end, the overall flow space is partitioned by FV into distinct classes, e.g., based on IP addresses and ports, and each guest is given control of a subset. Messages between switches and controllers are then filtered and translated accordingly.

We investigate an issue in where the switch from **Vendor C** works fine with the NOX controller, but not through the FV. We record the OpenFlow control plane traffic from the switch to FV in our production setup, as seen on the left side of Figure 5. We then replay the trace on a developer system, running *Ofreplay*, FV and the upstream controller on a single host for debugging. *Ofreplay* thus assumes the role of the switch.

Through repeated automated replay of the trace on the development host, we track down the source of the problem: It is triggered by a switch announcing a non-contiguous range of port numbers (e.g., 1, 3, 5). When FV translates a `FLOOD` action sent from the upstream controller to such a switch, it incorrectly expands the port range to a contiguous range, including ports that are not announced by the switch (e.g., 1, **2**, 3, **4**, 5). The switch then drops the invalid action.

Here, **OFRewind** proves useful in localizing the root cause for the failure. Replaying the problem in the development environment enables much faster turnaround times, and thus reduces debugging time. Moreover, it can be used to verify the software patch that fixes the defect.

4.5 NOX `PACKET-IN` Parsing Error

We now investigate a problem, reported on the NOX [31] development mailing list, where the NOX controller consistently drops the ARP reply packet from a specific host. The controller is running the `pyswitch` module.

The bug reporter provides a `tcpdump` of the traffic between their switch and the controller. We verify the existence of the bug by replaying the control traffic to our instance of the NOX. We then gradually increase the debug output from NOX as we play back the recorded OpenFlow messages to NOX.

Repeating this processes reveals the root cause of the problem: NOX deems the destination MAC address `00:26:55:da:3a:40` to be invalid. This is because the MAC address contains the byte `0x3a`, which happens to be the binary equivalent of the character ':' in ASCII. This "fake" ASCII character causes the MAC address parser to interpret the MAC address as ASCII, leading to a parsing error and the dropped packet. Here, *Ofreplay* provides the necessary debugging context to faithfully reproduce a bug encountered in a different deployment, and leads us to the erroneous line of code.

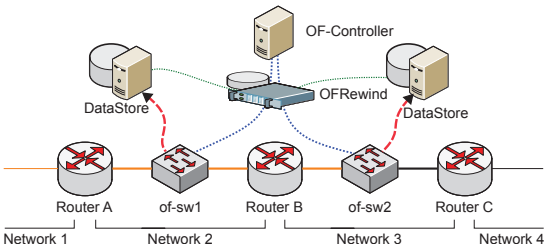


Figure 6: Quagga RIPv1 bug #235

4.6 Faulty Routing Advertisements

In a departure from OpenFlow network troubleshooting, we examine how **OFRewind** can be used to troubleshoot more general, event-driven network problems. We consider the common problem of a network suffering from a mis-configured or faulty router. In this case, we demonstrate how **OFRewind** can be advantageously used to identify the faulty component.

We apply **OFRewind** to troubleshoot a documented bug (Quagga Bugzilla #235) detected in a version of the RIPv1 implementation of the *Quagga* [34] software routing daemon. In the network topology given by Figure 6, a network operator notices that shortly after upgrading *Quagga* on software router B, router C subsequently loses connectivity to Network 1. As routing control plane messages are a good example of low-volume guest control plane traffic, they can be recorded by *Ofrecord* always-on or, alternatively, as a precautionary measure during upgrades. Enabling *flow creation sync markers* for the low-volume routing control plane messages ensures the global ordering is preserved.

The observation that router C loses its route to Network 1 while router B maintains its route, keys the operator to record traffic arriving at and departing from B. An analysis of the *Ofrecord* flow summaries reveals that although RIPv1 advertisements arrive at B from A, no advertisements leave B toward C. Host-based debugging of the RIPd process can then be used on router B in conjunction with *Ofreplay* to replay the trigger sequence and inspect the RIPd execution state. This reveals the root cause of the bug – routes toward Network 1 are not announced by router B due to this (0.99.9) version's handling of classful vs. CIDR IP network advertisements – an issue inherent to RIPv1 on classless networks.

4.7 Discussion

Without making any claims regarding the representativeness of the workload or switch behavior, in this limited space, we highlight in these case studies, the principle power and flexibility of **OFRewind**. We observe that **OFRewind** is capable of replaying subpopulations of control or data traffic, over a select network topology (switches and ports) or to select controllers, in a sandbox or production environment.

of-simple	reference controller emulating a learning switch
nox-pyswitch	NOX controller running Python `pyswitch` module
nox-switch	NOX controller running C-language `switch` module
flowvisor	Flowvisor controller, running a simple allow-all policy for a single guest controller
ofrecord	*Ofrecord* with substrate mode recording
ofrecord-data	*Ofrecord* with guest mode recording, with one data port and sync beacons and barriers enabled

Table 3: Notation of controllers used in evaluation

We further note that **OFRewind** has potential in switch (or controller) benchmarking. By creating a sandbox for experimentation that can be exported to a standard replay format, a network operator can concretely specify the desired behavior to switch (or controller) design engineers. The sandbox can then be run within the premises of the switch (or controller software) vendor on a completely new set of devices and ports. On receiving the device (or software), the network operator can conduct further benchmarking to compare performance of different solutions in a fair manner.

Comparison with traditional recording Based on the case presented in the last section, we compare the effort of recording and instrumenting the network with and without **OFRewind**. Note that while the specific traffic responsible for the failure is small (RIP control plane messages), the total traffic volume on the involved links may be substantial. To attain a synchronized recording of this setup without **OFRewind**, and in the absence of host-based instrumentation, one has to (1) deploy monitoring servers that can handle the *entire* traffic on each link of interest, e.g., [29], and redeploy as link interests change. Then, one must either (2a) reconfigure both switches to enable span ports (often limited to 2 on mid-range hardware) or (2b) introduce a tap into the physical wiring of the networks. Manually changing switch configurations runs a risk of operator error and introducing a tap induces downtime and is considered even riskier. (3) Additionally, the monitoring nodes may have to be synced to microsecond level to keep the flows globally ordered, requiring dedicated, expensive hardware. With **OFRewind**, one requires only a few commodity PCs acting as *DataStores*, and a single, central configuration change to record a *consistent* traffic trace.

5 Evaluation

When deploying **OFRewind** in a live production environment, we need to pay attention to its scalability, overhead and load on the switches. This section quantifies the general impact of deploying *Ofrecord* in a production network, and analyzes the replay accuracy of *Ofreplay* at higher flow rates.

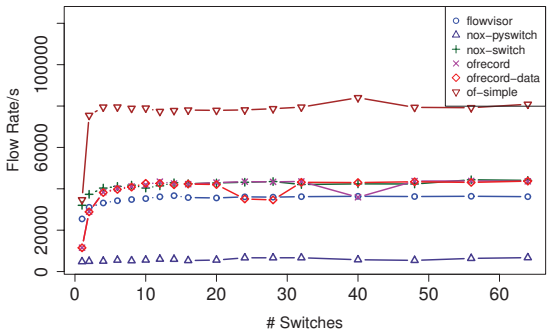


Figure 7: # Switches vs. median flow rate throughputs for different controllers using cbench.

5.1 *Ofrecord* Controller Performance

A key requirement for practical deployment of *Ofrecord* in a production environment is recording performance. It must record fast enough to prevent a performance bottleneck in the controller chain.

Using `cbench` [14], we compare the controller performance exhibited by several well known controllers, listed in Table 3. *Of-simple* and NOX are stand-alone controllers, while *flowvisor* and *ofrecord* act as a proxy to other controllers. *Ofrecord* is run twice: in substrate mode (*ofrecord*), recording the OpenFlow substrate traffic, and in guest mode (*ofrecord-data*), additionally performing OpenFlow message translations and outputting sync marker messages. Note that no actual guest traffic is involved in this experiment.

The experiment is performed on a single commodity server (Intel Xeon L5420, 2.5 GHz, 8 cores, 16 GB RAM, 4xSeagate SAS HDDs in a RAID 0 setup). We simulate, using `Cbench`, 1-64 switches connecting to the controller under test, and send back-to-back `PACKET-IN` messages to measure the maximum flow rate the controller can handle. `Cbench` reports the current flow rate once per second. We let each experiment run for 50 seconds, and remove the first and last 4 results for warmup and cool-down.

Figure 7 presents the results. We first compare the flow rates of the stand-alone controllers. *Nox-pyswitch* exhibits a median flow rate of *5,629* flows/s over all switches, *nox-switch* reports *42,233* flows/s, and *of-simple 78,908* flows/s. Consequently, we choose *of-simple* as the client controller for the proxy controllers. We next compare *flowvisor* and *ofrecord*. *Flowvisor* exhibits a median flow throughput of *35,595* flows/s. *Ofrecord* reports *42,380* flows/s, and *ofrecord-data* reports *41,743*. There is a slight variation in the performance of *ofrecord*, introduced by the I/O overhead. The minimum observed flow rates are *28,737* and *22,248*. We note that all controllers exhibit worse maximum throughput when only connected to a single switch, but show similar performance for $2 - 64$ switches.

We conclude that *Ofrecord*, while outperformed by *of-simple* in control plane performance, is unlikely to create a bottleneck in a typical OpenFlow network, which often includes a FlowVisor instance and guest domain controllers running more complex policies on top of NOX. Note that all controllers except *nox-pyswitch* perform an order of magnitude better than the maximum flow rates supported by current prototype OpenFlow hardware implementations (max. *1,000* flows/s).

5.2 Switch Performance During Record

When *Ofrecord* runs in *guest mode*, switches must handle an increase in OpenFlow control plane messages due to the sync markers. Additionally, `FLOW-MOD` and `PACKET-OUT` messages contain additional actions for mirroring data to the *DataStores*. This may influence the *flow arrival rate* that can be handled by a switch.

To investigate the impact of *Ofrecord* deployment on switch behavior, we use a test setup with two 8-core servers with 8 interfaces each, wired to two prototype OpenFlow hardware switches from **Vendor A** and **Vendor B**. We measure the supported flow arrival rates by generating minimum sized UDP packets with increasing destination port numbers in regular time intervals. Each packet thus creates a new flow entry. We record and count the packets at the sending and the receiving interfaces. Each run lasts for 60 seconds, then the UDP packet generation rate is increased.

Figure 8 presents the flow rates supported by the switches when controlled by *ofrecord*, *ofrecord-data*, and *of-simple*. We observe that all combinations of controllers and switches handle flow arrival rates of at least 200 flows/s. For higher flow rates, the **Vendor B** switch is CPU limited and the additional messages created by *Ofrecord* result in reduced flow rates (*ofrecord*: 247 flows/s, *ofrecord-data*: 187) when compared to *of-simple* (393 flows/s). **Vendor A** switch does not drop flows up to an ingress rate of 400 flows/s. However, it peaks at 872 flows/s for *ofrecord-data*, 972 flows/s for *ofrecord* and 996 flows/s for *of-simple*. This indicates that introducing *Ofrecord* imposes an acceptable performance penalty on the switches.

5.3 DataStore Scalability

Next we analyze the scalability of the *DataStores*. Note that *Ofrecord* is not limited to using a single *DataStore*. Indeed, the aggregate data plane traffic (T_s bytes in c_F flows) can be distributed onto as many *DataStores* as necessary, subject to the number of available switch ports. We denote the number of *DataStores* with n and enumerate each *DataStore* D_i subject to $0 \leq i < n$. The traffic volume assigned to each *DataStore* is T_i, such that $T_s = \sum T_i$. The flow count on each *DataStore* is c_i.

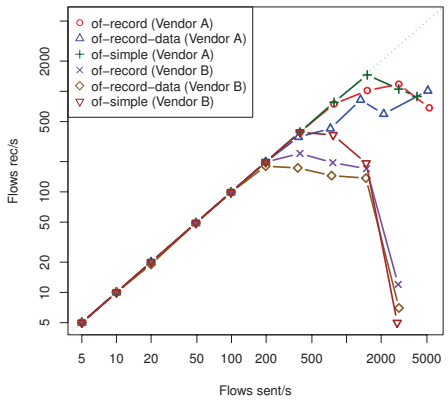


Figure 8: Mean rate of flows sent vs. successfully received with controllers *ofrecord*, *ofrecord-data*, and *ofsimple* and switches from **Vendor A** and **B**.

The main overhead when using *Ofrecord* is caused by the *sync markers* that are flooded to *all DataStores* at the same time. Thus, their number limits the scalability of the system. *Flow-sync markers* are minimum-sized Ethernet frames that add constant overhead ($\theta = 64B$) per new flow. Accordingly, the absolute overhead for each *DataStore* is: $\Theta_i = \theta \cdot c_i$. The absolute overhead for the entire system is $\Theta = \sum \Theta_i = \theta \cdot c_F$, the relative overhead is: $\Theta_{rel} = \frac{\Theta}{T_s}$.

In the Stanford production network, of four switches with one *DataStore* each, a 9 hour day period on a workday in July 2010 generated $c_F = 3,891,899$ OpenFlow messages that required synchronization. During that period, we observed 87.977 GB of data plane traffic. Thus, the overall relative overhead is $\Theta_{rel} = 1.13\%$, small enough to not impact the capacity, and allow scaling up the deployment to a larger number of *DataStores*.

5.4 End-to-End Reliability And Timing

We now investigate the end-to-end reliability and timing precision of **OFRewind** by combining *Ofrecord* and *Ofreplay*. We use minimum size flows consisting of single UDP packets sent out at a uniform rate to obtain a worst-case bound. We vary the flow rate to investigate scalability. For each run, we first record the traffic with *Ofrecord* in guest mode with flow sync markers enabled. Then, we play back the trace and analyze the end-to-end drop rate and timing accuracy. We use a two-server setup connected by a single switch of **Vendor B**. Table 4 summarizes the results. Flow rates up to 200 Flows/s are handled without drops. Due to the flow sync markers, no packet reorderings occur and all flows are replayed in the correct order. The exact inter-flow timings vary though, upwards from 50 Flows/s.

To investigate the timing accuracy further, we analyze the relative deviation from the expected inter-flow delay.

Rate (Flows/s)	Drop %	sd(timing, in ms)
5	0 %	4.5
10	0 %	15.6
20	0 %	21.1
50	0 %	23.4
100	0 %	10.9
200	0 %	13.9
400	19%	15.8
800	41 %	21.5

Table 4: **OFRewind**—end-to-end measurements with uniformly spaced flows consisting of 1 UDP packet

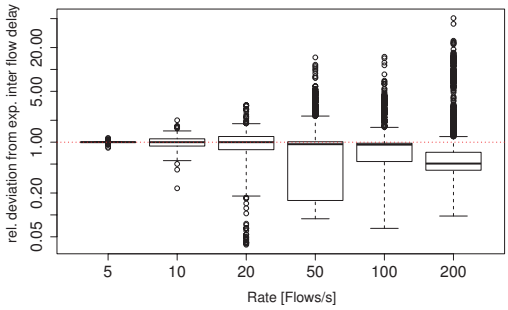


Figure 9: End-to-end flow time accuracy as a boxplot of the relative deviation from expected inter-flow delay.

Figure 9 presents the deviations experienced by the flows during the different phases of the experiment. Note that while there are certainly outliers for which the timing is far off, the *median* inter-flow delay remains close to the optimum for up to 100 Flows/s. Higher rates show room for improvement.

5.5 Scaling Further

We now discuss from a theoretical standpoint the limits of scalability intrinsic to the design of **OFRewind** when scaling beyond currently available production networks or testbeds. As with other OpenFlow-based systems, the performance of **OFRewind** is limited by the switch flow table size and the switch performance when updating and querying the flow table. We observe these to be the most common performance bottlenecks in OpenFlow setups today. Controller domain scalability is limited by the capacity of the link that carries the OpenFlow control channel, and the network and CPU performance of the controller. Specific to **OFRewind**, the control plane components require sufficient I/O performance to record the selected OpenFlow messages – not a problem in typical developments. When recording data plane network traffic, *DataStore* network and storage I/O capacity must be sufficient to handle the aggregate throughput of the selected flows. As load-balancing is performed over *DataStores* at flow granularity, **OFRewind** cannot fully record individual flows that surpass the network or storage I/O capacity of a single *DataStore*. When *flow creation markers* are used for synchronization, the overhead grows lin-

early with the number of *DataStores* and the number of flows. Thus, when the average flow size in a network is small, and synchronization is required, this may limit the scalability of a controller domain. For scaling further, **OFRewind** may in the future be extended to a *distributed controller domain*. While a quantitative evaluation is left for future study, we note that the lock-step approach taken to coordinate the replay of multiple instances of *Datarecord* and *Datareplay* (see Section 2.6) can be extended to synchronize multiple instances of **OFRewind** running as proxies to instances of a distributed controller. The same trade-offs between accuracy and performance apply here as well.

6 Related Work

Our work builds on a wealth of related work in the areas of recording and summarizing network traffic, replay debugging based on networks and on host primitives, automated problem diagnosis, pervasive tracing, and testing large-scale systems.

Recording/summarizing network traffic: Apart from the classical tool *tcpdump*[5], different approaches have been suggested in the literature to record high-volume network traffic, by performance optimization [12, 18], by recording abstractions of the network traffic [19, 27, 15], or omitting parts of the traffic [21, 26]. Our selection strategies borrow many of their ideas, more can be incorporated for improved performance. Note that these systems do not target network replay, and that all integrated control and data plane monitoring systems face scalability challenges when monitoring high-throughput links as the monitor has to consider all data plane traffic, even if only a subset is to be recorded. Similar to our approach of recording in a split-architecture environment, *OpenSafe* [13] leverages OpenFlow for flexible network monitoring but does not target replay or provide temporal consistency among multiple monitors. Complementary to our work, *OpenTM* [39] uses OpenFlow statistics to estimate the traffic matrix in a controller domain. *MeasuRouting* [37] enables flexible and optimized placement of traffic monitors with the help of OpenFlow, and could facilitate non-local *DataStores* in **OFRewind**.

Network replay debugging: *Tcpdump* and *tcpreplay* [6] are the closest siblings to our work that target network replay debugging. In fact, **OFRewind** uses these tools internally for data plane recording and replay, but significantly adds to their scope, scalability, and coherence: It records from a controller domain instead of a single network interface, can *select* traffic on the control plane and *load-balance* multiple *DataStores* for scalability, and can record a *temporally consistent* trace of the controller domain.

Replay debugging based on host primitives: Complementary to our network based replay, there exists a wealth of approaches that enable replay debugging for distributed systems on end-hosts [8, 17, 30]. DCR [9], a recent approach, emphasizes the importance of the control plane for debugging. These provide fully deterministic replay capabilities important for debugging complex end-host systems. They typically cannot be used for *black box* network components.

Automated problem diagnosis: A deployment of **OFRewind** can be complemented by a system that focuses on automated problem diagnosis. Sherlock diagnoses network problems based on passive monitoring [32], and other systems infer causality based on collected message traces [7, 38]. They target the debugging and profiling of individual applications while our purpose is to support debugging of networks.

Pervasive tracing: Some proposals integrate improved in-band diagnosis and tracing support directly into the Internet, e.g., by pervasively adding a trace ID to correlated requests [35] or by marking and remembering recently seen packets throughout the Internet [11]. We focus on the more controllable environment of a single administrative domain, providing replay support directly in the substrate, with no changes required to the network.

Testing large-scale networks: Many approaches experience scalability issues when dealing with large networks. The authors of [22] suggest to scale down large networks and map them to smaller virtualized testbeds, combining *time dilation* [23] and disk I/O simulation to enable accurate behavior. This idea may aid scaling replay testbeds for **OFRewind**.

7 Summary

This paper addresses an important void in debugging operational networks – scalable, economically feasible recording and replay capabilities. We present the design, implementation, and usage of **OFRewind**, a system capable of recording and replaying network events, motivated by our experiences troubleshooting network device and control plane anomalies. **OFRewind** provides control over the topology (choice of devices and their ports), timeline, and selection of traffic to be collected and then replayed in a particular debugging run. Using simple case studies, we highlight the potential of **OFRewind** for not only reproducing operational problems encountered in a production deployment but also localizing the network events that trigger the error. According to our evaluation, the framework is lightweight enough to be enabled per default in production networks.

Some challenges associated with network replay are still under investigation, including *improved timing accuracy*, *multi-instance synchronization*, and *online replay*. **OFRewind** can preserve flow order, and its timing is ac-

curate enough for many use cases. However, further improvements would widen its applicability. Furthermore, synchronization among multiple *Ofrecord* and *Ofreplay* instances is desirable, but nontrivial, and might require hardware support for accurate time-stamping [29].

In a possible extension of this work, *Ofrecord* and *Ofreplay* are combined to form an *online replay* mode. Recorded messages are directly replayed upon arrival, e.g., to a different set of hardware or to a different *substrate* slice. This allows for online investigation and troubleshooting of failures in the sense of a *Mirror VNet* [40].

Our next steps involve gaining further experience with more complex use cases. We plan to collect and maintain a standard set of traces that serve as input for automated regression tests, as well as benchmarks, for testing new network components. Thus, we expect **OFRewind** to play a major role in helping ongoing OpenFlow deployment projects[4] resolve production problems.

8 Acknowledgements

We wish to extend gratitude toward our shepherd, George Candea, for his guidance and help in shaping our final paper version, as well as to our anonymous reviewers for their remarkably detailed and insightful feedback, and to Deutsche Telekom Laboratories and the GLAB project for funding our work.

9 References

[1] EU Project Ofelia. `http://www.fp7-ofelia.eu/`.
[2] IETF Working Group Forces. `http://bit.ly/ieforces`.
[3] NEC Programmable Networking Solutions. `http://www.necam.com/PFlow/`.
[4] OFRewind Code. `bit.ly/ofrewind`.
[5] tcpdump. `http://www.tcpdump.org/`.
[6] tcpreplay. `http://tcpreplay.synfin.net/`.
[7] M. Aguilera et al. Performance debugging for distributed systems of black boxes. In *Proc. ACM SOSP*, 2003.
[8] G. Altekar and I. Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proc. ACM SOSP*, 2009.
[9] G. Altekar and I. Stoica. Dcr: Replay debugging for the datacenter. Technical Report UCB/EECS-2010-74, UC Berkeley, 2010.
[10] G. Altekar and I. Stoica. Focus replay debugging effort on the control plane. In *Proc. USENIX HotDep*, 2010.
[11] A. Anand and A. Akella. Netreplay: a new network primitive. In *Proc. HOTMETRICS*, 2009.
[12] E. Anderson and M. Arlitt. Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks. Technical Report HPL-2006-156, HP Labs, 2006.
[13] J. R. Ballard, I. Rae, and A. Akella. Extensible and scalable network monitoring using opensafe. In *Proc. INM/WREN*, 2010.
[14] CBench - Controller Benchmarker. `www.openflowswitch.org/wk/index.php/Oflops`.
[15] E. Cooke, A. Myrick, D. Rusek, and F. Jahanian. Resource-aware Multi-format Network Security Data Storage. In *Proc. SIGCOMM LSAD Workshop*, 2006.
[16] D. Geels et al. Replay debugging for distributed applications. In *Proc. USENIX ATC*, 2006.
[17] D. R. Hower et al. Two hardware-based approaches for deterministic multiprocessor replay. *Comm. ACM*, 52(6), 2009.
[18] P. Desnoyers and P. J. Shenoy. Hyperion: High Volume Stream Archival for Retrospective Querying. In *Proc. USENIX ATC*, 2007.
[19] F. Reiss et al. Enabling Real-Time Querying of Live and Historical Stream Data. In *Proc. Statistical & Scientific Database Management*, 2007.
[20] GENI: Global Environment for Network Innovations. `http://www.geni.net`.
[21] J. M. Gonzalez, V. Paxson, and N. Weaver. Shunting: A Hardware/Software Architecture for Flexible, High-performance Network Intrusion Prevention. In *Proc. 14th ACM CCS*, 2007.
[22] D. Gupta, K. Vishwanath, and A. Vahdat. Diecast: Testing distributed systems with an accurate scale model. In *Proc. USENIX NSDI*, 2008.
[23] Gupta, D. et al. To infinity and beyond: Time warped network emulation. In *Proc. ACM SOSP*, 2005.
[24] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating system with time-traveling virtual machines. In *Proc. USENIX ATC*, 2005.
[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.
[26] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *Proc. ACM SIGCOMM*, 2008.
[27] K. P. McGrath and J. Nelson. Monitoring & Forensic Analysis for Wireless Networks. In *Proc. Conf. on Internet Surveillance and Protection*, 2006.
[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *ACM CCR*, 38(2), 2008.
[29] J. Micheel, S. Donnelly, and I. Graham. Precision timestamping of network packets. In *Proc. ACM IMW*, 2001.
[30] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. ACM ASPLOS*, 2009.
[31] NOX - An OpenFlow Controller. `www.noxrepo.org`.
[32] P. Bahl et al. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proc. ACM SIGCOMM*, 2007.
[33] P. Phaal, S. Panchen, and N. McKee. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks, 2001.
[34] Quagga Routing Suite. `www.quagga.net`.
[35] R. Fonseca et al. X-trace: A pervasive network tracing framework. In *Proc. USENIX NSDI*, 2007.
[36] R. Sherwood et al. Carving Research Slices Out of Your Production Networks with OpenFlow. In *Proc. ACM SIGCOMM Demo Session*, 2009.
[37] S. Raza, G. Huang, C.-N. Chuah, S. Seetharaman, and J. Singh. MeasuRouting: A Framework for Routing Assisted Traffic Monitoring. In *Proc. IEEE INFOCOM*, 2010.
[38] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. Wap5: black-box performance debugging for wide-area systems. In *Proc. WWW*, 2006.
[39] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proc. PAM*, 2010.
[40] A. Wundsam, A. Mehmood, A. Feldmann, and O. Maennel. Network Troubleshooting with Mirror VNets. In *Proc. IEEE Globecom 2010 FutureNet-III workshop*, December 2010.
[41] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D Network Control Plane. In *Proc. USENIX NSDI*, 2007.

[4] There are ongoing production deployments of OpenFlow-enabled networks in Asia, Europe, as well as the US.

ORDER: Object centRic DEterministic Replay for Java

Zhemin Yang
Fudan University

Min Yang
Fudan University

Lvcai Xu
Fudan University

Haibo Chen
Fudan University

Binyu Zang
Fudan University

Abstract

Deterministic replay systems, which record and replay non-deterministic events during program execution, have many applications such as bug diagnosis, intrusion analysis and fault tolerance. It is well understood how to replay native (e.g., C) programs on multi-processors, while there is little work for concurrent java applications on multicore. State-of-the-art work for Java either assumes data-race free execution, or relies on static instrumentation, which leads to missing some necessary non-deterministic events.

This paper proposes the ORDER framework to record and reproduce non-deterministic events inside Java virtual machine (JVM). Based on observations of good locality at object level for threads and frequent object movements due to garbage collection, ORDER records and replays non-deterministic data accesses by logging and enforcing the order in which threads access objects. This essentially eliminates unnecessary dependencies introduced by address changes of objects during garbage collection and enjoys good locality as well as less contention, which may result in scalable performance on multicore. Further, by dynamically instrumenting Java code in the JVM compilation pipeline, ORDER naturally covers non-determinism in dynamically loaded classes.

We have implemented ORDER based on Apache Harmony. Evaluation on SPECjvm2008, PseudoJBB2005, and JRuby shows that ORDER only incurs 108% performance overhead on average and scales well on a 16-core Xeon testbed. Evaluation with a real-world application, JRuby, shows that several real-world concurrency bugs can be successfully reproduced.

1 Introduction

Deterministic replay has many applications such as diagnosing (concurrency) bugs [4, 8, 12, 18, 25, 26, 29], facilitating fault tolerance [2], forensic analysis [11] and offloading heavyweight dynamic program analysis [10]. Essentially, it works by recording non-deterministic events such as data access interleavings and interactions with external environments of a program during normal execution, and ensuring the same order of program execution by replaying the recorded events and enforcing constraints within the events. Currently, many deterministic replay systems for native code (e.g., C or C++ based programs) use a dependency-based approach that enforces the accessing order to a specific shared address at different granularities such as word [6], cache line [17, 22], or page [12, 16].

While the approaches to replaying native code have been studied extensively and relatively well understood, it is still unanswered question how to efficiently replay concurrent Java applications on multicore in a scalable and efficient way. Unlike native code, Java code usually needs to cooperate with the Java virtual machine (JVM) to achieve automatic garbage collection and to interact with the native code. Such runtime features introduce new non-determinism and more design considerations to implement a scalable and efficient deterministic replay system for Java code.

Many state-of-the-art deterministic replay systems for Java applications record *Logical Thread Schedule* [28, 9, 30], which assumes that applications are running on uni-processor platforms. Such a strategy is unsuitable for replaying concurrent Java applications running on multi-processor platforms. JaRec [13] records non-determinism in lock acquisition, but cannot reproduce buggy execution caused by data race. LEAP [15] uses static instrumentation for Java code to replay interleaved data accesses, thus it cannot reproduce non-determinism introduced by external code, such as libraries or class files dynamically loaded during runtime. Furthermore, LEAP does not distinguish different instances of the same class, and false dependencies between different objects of the same class may lead to large performance overhead when a class is massively instantiated.

Record time and log size are two critical performance metrics for deterministic replay systems, which can typ-

ically be optimized by applying transitive reduction in dependency-based address tracking approaches [6, 12, 17, 23]. However, these techniques may not be suitable for Java applications, due to frequent object movements by garbage collector. According to our evaluation results, with word or cache-line level address tracking approaches, garbage collection of JVM will introduce 7 times more unnecessary dependencies for SPECjvm2008 and SpecJBB2005. Furthermore, many Java programs have good locality on accessing a single object for Java threads.

Based on the observation above, this paper proposes ORDER, *Object centRic DEterministic Replay*, to identify data access dependencies at object granularity. Such an object-centric technique can avoid recording massive unnecessary dependencies introduced by object movements from garbage collector, reduce contention on accessing shared metadata due to the low probability of object-level interleavings, and improve recording locality by inlining shared-memory access information within object headers. By dynamically instrumenting Java code during JVM compilation pipeline, ORDER naturally covers non-determinism caused by dynamically loaded classes and libraries.

We have implemented ORDER based on Apache Harmony, to record and replay non-deterministic events for concurrent Java applications on multicore. To further improve the performance of ORDER, we have also implemented a compiler analysis algorithm based on Soot [31] to avoid tracking dependencies for thread-local and assigned-once objects. Besides, we implement an offline log compressor algorithm to filter out remaining unnecessary dependencies from thread-local and assigned-once objects caused by imprecise compiler analysis.

Performance evaluation results show that ORDER has relatively good and scalable performance on a 16-core Intel machine for SPECjvm2008, PseudoJBB2005, and JRuby. The average overhead for recording non-determinism is around 108%. ORDER is also with good scalability on a 16-core platform. Performance comparison with LEAP [15] shows that ORDER is 1.4X to 3.2X faster than LEAP. We also show that ORDER can successfully reproduce several real-world concurrency bugs in JRuby.

In summary, the contribution of this paper includes:

- Two observations (i.e., GC-introduced dependencies and object access locality) for deterministically replaying Java applications based on a study of Java runtime behavior.
- The case for *object-centric deterministic replay*, which leverages the object granularity to record non-deterministic data access events using dynamic instrumentation.
- The implementation and evaluation of ORDER based on a real-world JVM platform, which demonstrate the efficiency and effectiveness of ORDER.

This paper is organized as following. In next section, we will present our study with evaluation results on the runtime behavior of Java programs on multicore platforms. In Section 3, we describe the main idea and design of our object-centric deterministic replay approach. The implementation details of our prototype ORDER are presented in Section 4. Section 5 shows the evaluation results in terms of performance, scalability, log size and bug reproducibility of ORDER. Finally, section 6 summarizes related work in deterministic replay and section 7 concludes our work with a brief overview of possible future work.

2 Java Runtime Behavior

In Java runtime environment, *garbage collection (GC)* is commonly used to automatically reclaim *non-reachable* memory spaces. The use of GC enables automatic memory management and avoids many memory-related bugs such as *dangling pointers*, *double free*, and *memory leakage*. GC usually requires moving or modifying objects in heap, which may cause additional dependencies for deterministic replay. In this section, we evaluate the impact of GC and describe two major observations that may affect the scalability and performance of deterministic replay systems.

2.1 Environment Setup and Workloads

The experimental results listed below are all generated on a machine with 4 quad-core Xeon processors (1.6GHz) and 32 GB physical memory. The Linux kernel version is 2.6.26 and the version of Apache Harmony is m12. We evaluate 21 parallel Java applications from *SPECjvm2008*, *SPECjbb2005*, and *JRuby*. *SPECjvm2008* is a general-purpose benchmark suite composed of a number of multithreaded applications. We omit the result for *sunflow* because it failed to be compiled by Apache Harmony m12 on our evaluation platform. *SPECjbb2005* is a server-side Java application that simulates an online marketing system. It emulates a common 3-tier system, and focuses on the business logic and object manipulation. *JRuby* is a Java implementation of the Ruby Programming language and provides a Ruby Interpreter entirely written in Java.

Each benchmark of *SPECjvm2008* is configured to run a single iteration, which ensures a fixed workload. Results of JRuby are collected on the most recent stable version of JRuby (JRuby 1.6.0) and a multi-threaded Ruby benchmark provided by it (i.e., *bench_threaded_reverse*). If the number of threads is not specifically mentioned, all results are collected with 16 threads. All tests are tested three times and we report the average of them.

2.2 Dependency-based Replay

Many dependency-based deterministic replay techniques record data dependencies according to data addresses. They record data dependencies when two instructions access the same address [6], cache line [17], or page [12, 16]. *Dependencies, conflicts or constraints* [6, 12, 17, 23] *a -> b* in these systems indicate that 1) instruction *a* and *b* both access the same memory location; 2) at least one of them is a write; and 3) *a* happens before *b*. To make execution deterministic, a replay run ensures that *b* does not happen until *a* has been executed.

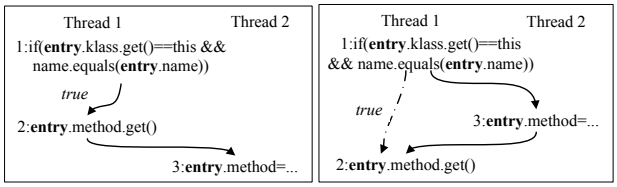


Figure 1: A real-world concurrency bug reported in JRuby community (JRuby-1382). Application crashes when statement 3 is executed between statements 1 and 2.

Figure 1 shows a real-world bug in JRuby. In this example, JRuby maintains a method cache. Thread 1 checks whether the required method resides in the cache. In correct execution (1 to 2 to 3), after the comparison code (statement 1) returns true, JRuby obtains the corresponding method from the cache, which is supposed to be the required method. However, if the content of the global variable "entry" is modified between statements 1 and 2 (1 to 3 to 2), the method obtained in statement 2 may be unexpected and crash the program. Suppose we treat reads/writes to each object as accesses to the same memory location, then there are two data conflicts in this example, 1,3 and 2,3. Dependency-based systems record the order of conflicted data instead of the order of all executed statements. For example, in Figure 1, if the application executes correctly, (2->3) is recorded. In buggy execution, (1->3) and (3->2) is recorded.

The number of recorded dependencies relies on the granularity chosen by deterministic replay systems. For example, if a deterministic replay system traces data dependencies according to the *real address* of data, dependency (1->3) in the given example may not be recorded. Specifically, statement 1 and 3 do not access the same memory location of data because *entry.klass* and *entry.name* have a different address from *entry.method*. Likewise, whether dependency (1->3) is recorded in page-level dependency-based replay relies on whether *entry.klass* or *entry.name* resides in the same page as *entry.method*. Though the granularity of recording dependency does not affect the correctness of a replay strategy, large performance overhead will be introduced if it is either too small(large disk operations), or too large (false sharing).

Instead of directly recording dependencies, BugNet [22] and PinPlay [27] log the value of load instruction after another thread modifies the same location. In these deterministic replay systems, the number of logged values depends on the number of conflicts occurred in the execution of programs. Although their logging approaches are different from recording dependencies, their performance is also affected by the Java runtime behavior we list below.

2.3 Observation 1: Dependencies from GC

In JVM, GC is triggered if the memory management scheme indicates that performing GC is beneficial. Each time GC is triggered, it will scan the entire heap space, mark the reachable objects, remove non-reachable objects from heap, and possibly move reachable objects to achieve better cache locality and fewer heap fragments. Both marking and moving reachable objects will introduce a large amount of write operations in the heap. Thus, when using data addresses or cache-lines to identify dependencies in Java applications, there are a lot of extra dependencies introduced by GC. Most of these dependencies do not truly affect the behavior of Java applications, thus they are not necessary to be recorded. Furthermore, the dependency boost will cause a long pause time in GC and deteriorate application performance.

Figure 2 shows the ratio of dependencies generated by two widely-used garbage collectors to those generated by an application itself. In Apache Harmony, when using the popular generational garbage collector, dependencies introduced by garbage collection are about 8 times the dependencies introduced by original Java applications. The dependency boost even exceeds 16 times the dependencies introduced by original application in *scimark.sor.large*, *xml.validation*, *scimark.fft.small*, and *SPECjbb2005*. Results of Mark-Sweep garbage collector are similar to those of Generational GC, which indicates that such a phenomenon is likely to be a common case.

GC itself is a non-deterministic event in the JVM. Object allocation order, total heap size, garbage collection algorithm, and many other runtime events will affect the behavior of GC. Thus, deterministic replay system cannot ignore the influence from GC. However, recording garbage collection behavior may cost much and worsen program performance.

2.4 Observation 2: Interleavings of Object Accesses

Within JVM, *object*, a new candidate of granularity for recording dependencies, is introduced by the managed

memory strategy. According to our experiments, objects accessed by a thread are very likely to be accessed by the same thread at the next time. Hence, interleavings[1] seldom happen at object level. As depicted in Table 1, the ratio of interleavings at object level is less than 7%. This ratio is extremely low in scientific applications (*fft*, *lu*, *sor*, *sparse*, *monte_carlo*). This implies that recording dependencies among threads at object level will likely result in better locality and less contention.

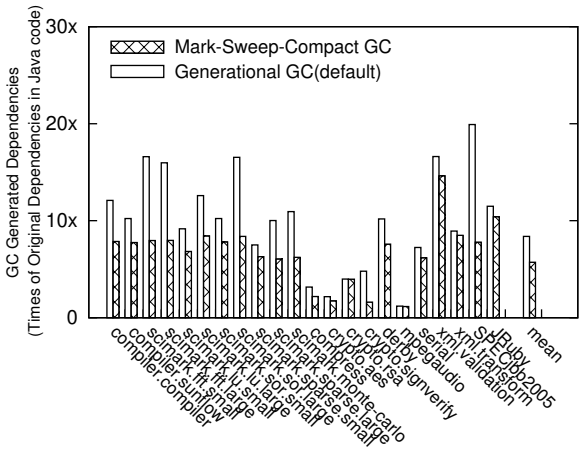


Figure 2: Dependencies introduced by GC when using data address to identify dependencies in Java applications. The base line is the number of dependencies introduced by Java applications themselves. Number of dependencies is calculated according to the CREW protocol in SMP-Revirt [12].

Case	Interleaving	Access	Rate(%)
compiler.compiler	53997073	3.7E+9	1.46
compiler.sunflow	159104781	7.6E+9	2.09
fft.small	6281	1.2E+10	<0.01
fft.large	3447	1.6E+10	<0.01
lu.small	6500	3.4E+10	<0.01
lu.large	3311	2.87E+11	<0.01
sor.small	4446	2.5E+10	<0.01
sor.large	3358	1.0E+11	<0.01
sparse.small	4201	3.0E+10	<0.01
sparse.large	3055	1.1E+11	<0.01
monte_carlo	3503	9.6E+10	<0.01
compress	448683851	3.4E+10	1.31
crypto.aes	3.73E+9	6.0E+10	6.21
crypto.rsa	135072884	2.2E+10	0.62
crypto.signverify	33185584	2.3E+10	0.14
derby	2.44E+9	4.9E+10	4.95
mpegaudio	922855001	6.4E+10	1.45
serial	315661230	1.7E+10	1.80
xml.validation	96681920	6.3E+9	1.53
xml.transform	1.41E+9	6.6E+10	2.13
SPECjbb2005	78856923	1.9E+15	<0.01
JRuby	161801036	1.3E+12	0.01

Table 1: Ratio of interleavings at object level: the second and third column show the number of interleavings and total object accesses accordingly. The forth column shows the percentage of interleavings among the total number of object accesses.

3 Object-centric Deterministic Replay

Based on the two observations, we propose an Object centRic DEterministic Replay (ORDER) scheme to record and replay concurrent Java applications. ORDER uses object as the granularity to record interleavings of data accesses. In the rest of this section, we first discuss why object would be a proper granularity of tracking dependencies, and then illustrate the sources of non-determinism within JVM and how ORDER handles them.

3.1 Why Object Centric?

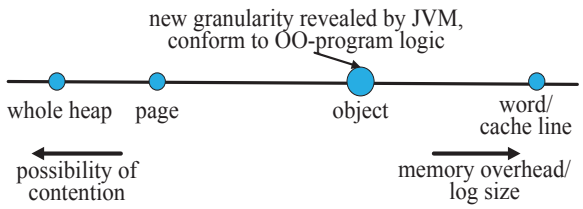


Figure 3: Spectrum of granularity in deterministic replaying Java applications.

In object-oriented programming languages like Java, applications are usually designed around objects. Figure 3 shows the spectrum of design consideration on the granularity of deterministic replay for Java. This implies several advantages of ORDER:

Elimination of GC dependencies: Our first observation above shows that massive extra dependencies will be raised by GC if tracking dependencies using addresses. However, such extra dependencies will naturally be eliminated when tracking dependencies at object level, as the movement of an object does not change its content.

Reduced contention of synchronization: When recording data dependencies, data content should be protected through synchronization to avoid possible data races. Our second observation indicates an object will likely be accessed consecutively by one thread. This implies less contention over the synchronization construct protecting the metadata information of an object.

Improved locality: Furthermore, Java does not support pointer arithmetic, and the memory layout of Java applications is managed by JVM. By embedding metadata information into object headers, there will be good locality for accessing such metadata information.

3.2 Recording Data Access Interleavings

As discussed in previous work [4, 8, 12, 18, 25, 26, 29], many bugs introduced by non-deterministic events in multi-threaded applications are caused by concurrent data accesses, i.e., the order that different threads access the same data. In Java applications, data are usually grouped as objects. Thus, access to a memory location can be considered as access to the corresponding object. This is a major source of non-determinism.

Instead of recording conflict access pairs, ORDER only records the object access timeline. The recording/replaying scheme in ORDER is depicted in Figure 4. ORDER records how many times a thread has accessed an object before this object is accessed by another thread. ORDER maintains such access timeline and enforces it during a replay run.

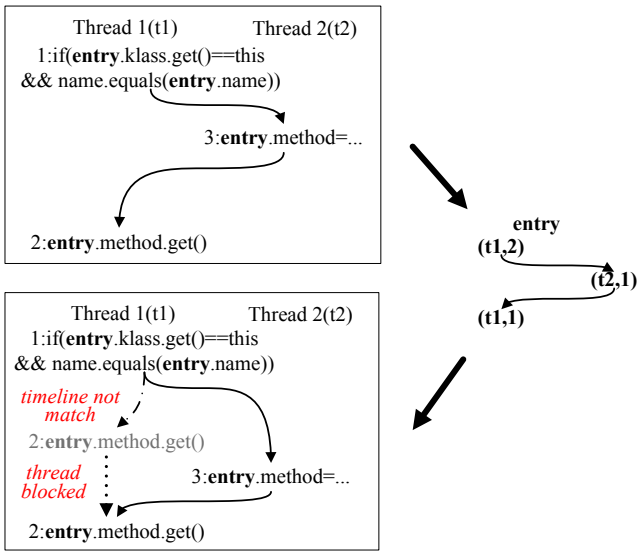


Figure 4: The recording/replaying scheme in ORDER. Each tuple <tn, x> represents a timeline record, which indicates that the object is accessed by thread tn for x times.

3.2.1 Metadata

We extend the header of each object with five fields to record the timeline of object accesses:

Object identifier (OI): Information in the original object header cannot uniquely identify an object. ORDER extends the original 32-byte hash-code to 64-byte so that it can uniquely identify an object in long-term execution. This new identifier is generated according to the identifier of the allocating thread and the index that indicates the object allocation order in this thread.

Accessing thread identifier (AT) and access counter (AC): Accessing thread identifier and access counter maintain current status of object access timeline. Every timeline recorded by ORDER can be interpreted as "*this object (OI) is accessed by some thread (AT) for some times (AC)*". In record phase, these two fields record: (1) which thread is now accessing this object; (2) how many times this thread has accessed it. In replay phase, they maintain: (1) which thread is expected to access this object; (2) how many times this thread will further access such an object before an expected interleaving is encountered.

Object-level lock: When recording/replaying object accesses, the accessed content as well as access thread (AT)/access counter (AC) should be synchronized. ORDER uses an object-level lock to protect the whole object (including fields, array elements and object header) when a certain access to this object is recorded/replayed. Using object granularity, our approach only needs to synchronize the accessed object instead of the whole heap or page, which may reduce the strength of synchronization.

Read-Write flag: The Read-Write flag records whether the current timeline record is *read-only* or *read-write*. This information is used in the timeline filter to reduce log size.

3.2.2 Recording/Replaying Object Access Timelines

ORDER contains two modes corresponding to the record and replay phase of deterministic replay systems respectively. Each mode contains an instrumentation action added to compilation pipeline of JVM. We also modify garbage collector to record the final state of timelines.

Record mode: Figure 5 illustrates how ORDER records object access timeline. In record mode, when an object is about to be accessed by a certain thread, ORDER compares AT in object header with the identifier of current accessing thread (CTID). If this access is a continuous access (AT == CTID) (Figure 5.a), ORDER updates the access counter (AC = AC + 1). When an interleaving is encountered (AT != CTID) (Figure 5.b and 5.c), ORDER puts the timeline record to log and resets timeline record in the object header (AT = CTID, AC = 1). When JVM is terminated, or objects are collected by GC, ORDER records the final timeline record of each object to log (Figure 5.d). Besides, the record operations as well as the original object access are enclosed by an object-level lock acquire/release pair, which ensures the atomicity of the record process. The Read-Write flag is set to *read-write* if a write operation is encountered.

Replay mode: Figure 6 illustrates how ORDER reproduces the recorded timeline. In replay mode, when ORDER is about to reproduce timeline record for a certain object, it loads the timeline record (AT and AC) into the object header. When a thread is about to access this object, the code instrumented by ORDER will compare its identifier (CTID) with expected thread identifier (AT) in object header. If the requesting thread is the expected ac-

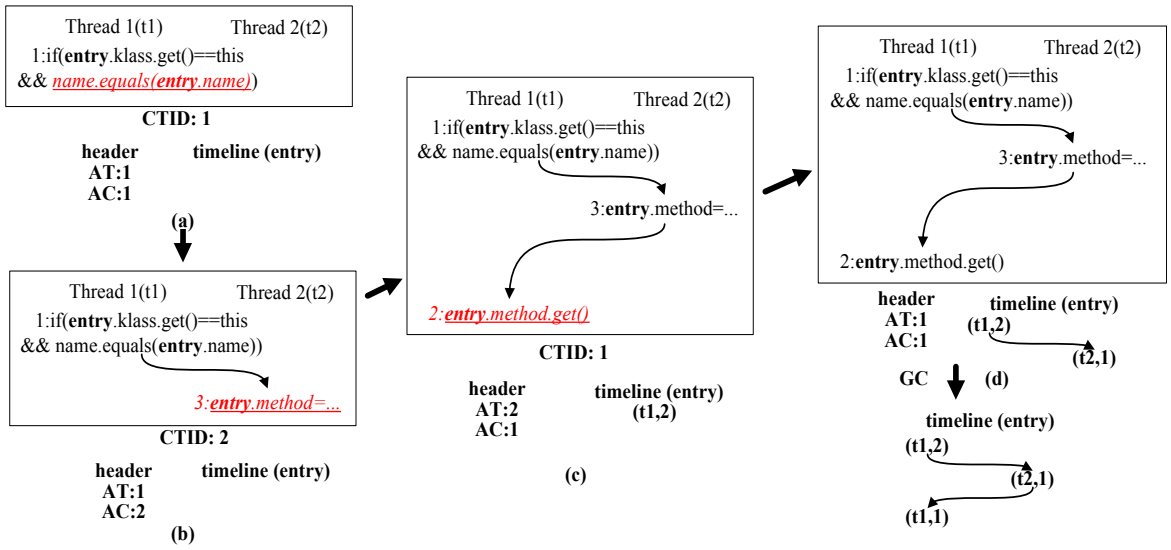


Figure 5: Record mode of ORDER.

cess thread (CTID == AT) (Figure 6.a, 6.b), ORDER updates the access counter (AC = AC - 1). When a recorded interleaving is about to occur (AC == 0), ORDER loads the next timeline record into the object header. If a requesting thread is not the expected access thread (CTID != AT), it will be blocked until a recorded interleaving updates the timeline record (Figure 6.c). If the blocked thread no longer violates the recorded timeline, it will be woken up and continue its execution (Figure 6.d). Like the record mode, a similar object-level lock is used to protect the whole object when certain access to this object is replayed.

3.2.3 Eliminating Unnecessary Timeline Records

Though object-level locks cause less contention than page/heap-level locks, recording object access timelines still incurs notable performance slowdown. According to our study, much of the recording overhead comes from instrumentation to the following objects, which never cause non-determinism:

Thread-local objects: Many objects allocated by JVM are thread-local objects. These objects are accessed in a certain thread and never shared with other threads.

Assigned-once objects: Assigned-once objects have continuous write operations in their initialization methods. After initialization, the assigned-once objects are shared among different threads but no thread will write the fields of these objects. Such objects do not really introduce non-determinism to Java applications. According to our evaluation, assigned-once objects are very common in the JVM. For example, the switch table objects generated by Javac, class objects, string objects, and most of the final arrays are assigned-once objects.

To eliminate such unnecessary recording of object access timelines, an offline preparation phase is introduced to analyze the target application and annotate the Java bytecode. Recording/replaying phase of ORDER can adopt annotations from bytecode and eliminate unnecessary recordings.

Accesses to these two kinds of objects can be identified by inter-procedure analysis. We use escape analysis [7] to find thread-local objects. The original escape analysis algorithm introduces three escape states. Each state represents a certain kind of object: *NoEscape* means that objects allocated by a certain allocation site are method-local objects, which do not escape outside this method. *ArgEscape* represents the objects that are exposed to other functions, but they are not visible to different threads. *GlobalEscape* means that objects are shared among threads. However, because *GlobalEscape* does not contain read/write information, the original escape analysis cannot identify assigned-once objects. In ORDER, we apply the following modifications to the original algorithm so that assigned-once objects can be identified:

- A new escape state *Shared-Write* is introduced. Object nodes in the connection graph are set to such a state when: 1) Objects allocated by the corresponding new site can be global escaped; 2) Write operations may be applied to these objects after they are shared among threads.

- The read/write states of phantom object nodes, which represent the reference to object nodes that are not allocated in the local method, are traced. It can be *Read-Only* or *Read-Write* which means whether the corresponding objects can be written

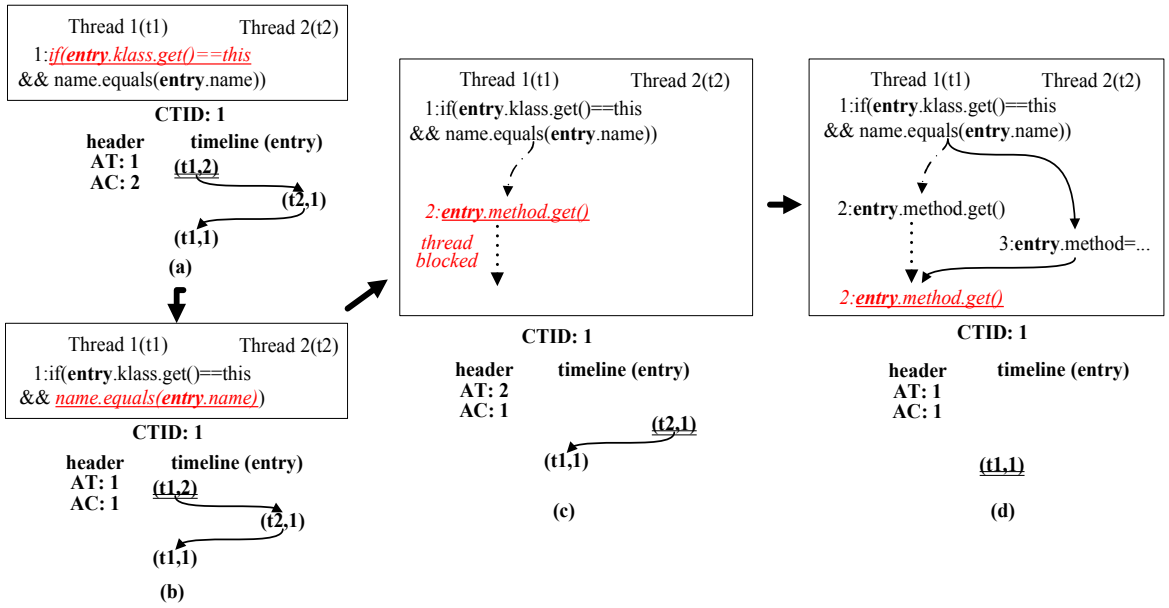


Figure 6: Replay mode of ORDER.

in the current method. When a new phantom node is created, its read/write state is set to *Read-Only*. In intra-procedure analysis, if a write operation is applied to p, the read/write states of all phantom object nodes pointed by p are switched to *Read-Write*. Because read/write state is only attached to phantom object nodes, they are finally marked as assigned-once objects if their read/write states are *Read-Write*.

- The read/write information is used to identify assigned-once objects. It affects the escape state in the following way. First, we modify the transfer function of escape state in intra-procedure analysis that if a certain node is already *Global-Escape* and a write operation is encountered, its escape state is set to *Shared-Write*. Second, in the inter-procedure merge function, when a *Read-Write* phantom object node is merged to a *Global-Escape* normal object node, which means that globally shared objects can be modified outside the allocation function, the escape state of this normal object node is changed to *Shared-Write*.

To avoid eliminating necessary timeline records, our offline analysis algorithm is conservative. Specifically, when an object node might possibly be exposed to external code, its escape state is automatically set to *Shared-Write*.

3.2.4 Log Compression

The raw log recorded by ORDER contains object access timelines of all recorded objects. Although inter-procedure escape analysis can eliminate some unnecessary timeline records, ORDER still records many thread-local or assigned-once objects due to the imprecision of the analysis algorithm. Timeline records of such objects does not help to correct deterministic replay in the replay phase. Thus, we apply a timeline filter to eliminate these unnecessary object access timelines. The timeline filter analyzes the timeline information in the log and filters out the following ones: a) Timeline that has only one occurrence; b) Timeline that has only one occurrence of read-write in the beginning. Besides, timelines which have several occurrences of read-write in the beginning are partially eliminated. We reserve the leading read-write occurrences and eliminate the following read-only ones. The log compressor costs short time and can be processed by an idle core, and it can be applied either offline or online in GC. Currently, ORDER applies it offline and we plan to apply this filter online in the future.

3.3 Other Non-determinism in JVM

Some sources of non-determinism are common for both Java and native code, while the JVM additionally introduces sources of non-determinism such as garbage collection, adaptive compilation and class initialization. In the following, we describe each source of non-determinism and how ORDER handles it.

3.3.1 Common Non-determinism in Java and Native Code

Lock acquisition: When two threads are competing on the same lock, the order of lock acquisition is an important source of non-determinism. Thus, it is neces-

sary to record/replay the lock acquisition/release order. The Java programming language provides two basic synchronization idioms: synchronized methods and synchronized statements, each of which is ensured by maintaining a corresponding critical section in the JVM. In ORDER, entering a critical section is treated as an access to the corresponding object. Besides, JVM also provides explicit locks and atomic operations in package *java.util.concurrent*. Likewise, ORDER treats these operations as accesses to the lock objects.

Signal: Similar to C/C++ programs, signals also cause non-deterministic behavior in Java applications. In JVM, they are usually wrapped to wait, notify, and interrupt operations for threads. Leveraging their non-preemptive nature, ORDER records the return values and status of the pending queue instead of the triggering time. Java developers can write preemptive signal handlers by using the "sun.misc.SignalHandler" interface, though they are neither officially supported according to Sun, nor supported by many state-of-the-art Java Virtual Machines. By tracing and recording the timeline status of the current active object, which is the last accessed object in the current thread, non-determinism in such libraries can also be reproduced by ORDER.

Program Input: Different input may generate different program behavior. ORDER records this non-determinism by logging the content of input to Java programs.

Library invocation : Some methods of the Java library, like *System.getCurrentTimeMillis()* and methods in *Random*/*SecureRandom* classes, generate non-deterministic return values. ORDER logs the return values of these methods to ensure determinism. Besides, non-deterministic events in native libraries are exposed to Java applications through lock acquisition, signal, input, garbage collection, class initialization, and the non-deterministic Java libraries. They are handled in ORDER as we covered in the corresponding sources of non-determinism.

Configuration of OS/JVM : To ensure that the environment setting of a record run is consistent with that of a replay run, ORDER records the configuration of OS/JVM and reproduces the recorded configuration in the replay run.

3.3.2 Unique Non-determinism in JVM

Garbage collection: Garbage collection is another source of non-determinism in the JVM. In multi-threaded Java applications, different order of object allocations across threads may cause different heap layout in memory, which then causes different collector behavior. As a result of the object-centric recording of data accesses, ORDER does not need to record dependencies introduced by GC. However, GC can affect the behavior of Java applications through several interfaces. To record such non-determinism, ORDER logs interfaces between GC threads and Java threads. Since most interface invocations are triggered by JVM, non-determinism should be recorded according to the JVM inner mechanism. We will discuss the detail later in section 4.

Adaptive Compilation : Adaptive compilation, which is also known as incremental compilation, recompiles methods if they are frequently invoked. Behavior of adaptive compilation relies on the profiling result of program execution. Because the profiling result varies in different executions, the behavior of adaptive compilation is also non-deterministic. Reproducing non-determinism caused by adaptive compilation can be supported by recording virtual machine states and profiling results introduced by Ogata et al. [24].

Class Initialization : When JVM resolves and initializes a class, static fields of this class are commonly initialized by the thread resolving the class. Thus, which thread first invokes the class resolution method may affect the behavior of Java applications. ORDER records the resolution and initialization thread identifier, and ensures that the same thread first enters the class resolution method in the replay run.

3.4 Discussions

Coverage of non-determinism: To our knowledge, ORDER is the first deterministic replay system which records non-determinism introduced by the Java runtime, such as GC, class initialization, etc. Moreover, unlike JaRec and LEAP, ORDER not only captures non-deterministic lock acquisition or data access interleaving outside JVM library, but also records non-deterministic events inside the library. Although recording such non-deterministic events incurs additional runtime overhead, we believe they are necessary and essential to deterministic replay systems. As discussed in LEAP, some bugs may not be reproducible due to the ignoring of these non-deterministic events. More importantly, loss of such non-determinism may unexpectedly deadlock normal program execution due to the inconsistent execution between recorded execution and replaying one. When replaying long-running Java applications, the replay system may deadlock itself before the buggy instruction is encountered.

Transitive log reduction: The timeline recorded by ORDER is already the smallest set of object access interleavings, which does not need to be further optimized by transitivity reduction [23]. Currently, ORDER does not separate conflicts from read-read dependencies. Our evaluation results in section 5 show that tracing timeline incurs much more overhead than swapping logs to disk. Though identifying conflicts from read-read dependencies can further reduce log swapping overhead, which

is already very small, it will notably increase the complexity of timeline tracing logic. Thus, ORDER does not apply complex conflict-based reduction algorithm like SMP-Revirt [12].

4 ORDER Implementation

We have implemented a prototype of ORDER based on Apache Harmony [1]. We add several new instrumentation phases into Harmony compilation pipeline to support deterministic replay. Besides, the default garbage collector of Harmony (Generational GC with default configuration for each object space) is modified to record final timeline of each object. We also modify Soot[31] to annotate thread-local or assigned-once object accesses in methods. Such annotation is attached to Java bytecode as a new attribute, which can be simply discarded if the target JVM does not support ORDER. Thus, it does not affect the portability of original Java application. Because preemptive handlers are currently not implemented in Apache Harmony, the current version of ORDER does not handle the corresponding non-determinism. In the following sections, we discuss how ORDER records non-determinism and cooperates with native code in Harmony.

4.1 Modification to Harmony Compilation Pipeline

Harmony uses *pipelines* to manage compilation configuration of Java methods. In Harmony, every *pipeline* contains a set of *actions* each of which represents a single analysis or optimization of Java methods. The instrumentation processes of record and replay phase are implemented as two *actions* separately in ORDER. Besides, if adaptive compilation is enabled, two or more *pipelines* can be assigned to a single method. Then, when a method is frequently invoked, it can be recompiled with a more aggressive *pipeline*. Whether adaptive compilation is enabled or not only affects the performance of Java application and does not affect the bug reproducibility of ORDER. Thus, the current prototype of ORDER disables adaptive compilation to reduce engineering effort and uses a single compilation pipeline. Type/copy propagation, constant folding, dead/unreachable code elimination, devirtualization and all platform dependent optimizations except peephole and fast array filling [3] are enabled in the selected pipeline.

4.2 Recording GC in Harmony

Although garbage collection is an important source of non-determinism in the JVM, it rarely affects the behavior of Java applications. ORDER does not record garbage collection activity in Harmony, but only records the following interfaces between garbage collection threads and Java threads:

1. After garbage collection, dead objects that have finalization methods should be finalized. The order that the finalization methods are invoked depends on heap layout and garbage collection algorithm. In Harmony, finalizable objects are enqueued to a specific object queue after garbage collection. Finalizing threads extract objects from the queue and invoke their *finalize* methods. ORDER records the order they are extracted from the queue and reproduces the recorded order in replay run.

2. In Java, weak/soft/phantom reference represents several strengths of ”non-strong” object instances, and they are collected in GC according to the memory usage. After GC, JVM notifies the queue of weak reference objects that the status of corresponding weak objects may be changed by garbage collection. Likewise, queues of soft/phantom reference objects are notified in the same way. The size of weak/soft/phantom reference set depends on runtime heap status. Like finalization, Harmony maintains a *references_to_enqueue* queue to link the *reference_enqueue* thread with Java threads. The order these objects are extracted from *references_to_enqueue* queue is recorded.

3. Java applications can explicitly invoke method *Runtime.freeMemory* to query the size of free memory from JVM. Different heap layouts result in different free memory sizes during execution. Because this method has no side effect, ORDER only records the return value of it.

4.3 Cooperating with JVM Native Code

ORDER uses dynamic instrumentation to guarantee that the replay run has the same object access timeline as the recorded run has. Although execution behavior of Java code is deterministic with the help of instrumentation, the execution behavior of JVM native code is still non-deterministic. ORDER records non-deterministic events of JVM native code that will cause non-deterministic behavior of Java applications, and ignores those not really affecting Java applications. ORDER should cooperate properly with ignored JVM native code so that the inconsistency between them will not introduce deadlock.

The internal suspend-resume mechanism of Harmony does not affect application behavior. Thus, it is ignored by ORDER. In Harmony, when enumerating the root set of the Java heap, GC threads suspend Java threads in order to get a consistent snapshot of the Java heap. When Java threads are about to enter *enable-suspend* state (named safe-point or safe-region), it must record the status of the current stack frame so that GC threads can obtain a complete set of live objects. However, frequently recording stack frame information is costly

and worsens performance. In Harmony, safe-point/safe-region is only invoked at call sites or certain system calls (e.g., sleep). ORDER blocks Java threads when they are about to violate the recorded object access timeline. When the blocked time of a Java thread is beyond a threshold (500ms), it prepares its own stack frame information and enters the *enable-suspend* state. When a Java thread enters the *enable-suspend* state, it can be easily suspended by other threads. The Java thread exits the enable-suspend state when it does not violate recorded timeline further.

5 Evaluation Results

In this section, we evaluate the performance slowdown of ORDER. We use an Intel Xeon machine with 4 quad-core 1.6Ghz CPUs and 32 GB physical memory, which runs a Linux with kernel version 2.6.26. We show the results for the SPECjvm2008 suite except *sunflow*, *derby* and *xml.transform*. One of them, *sunflow*, failed to be compiled by original Apache Harmony m12, and two of them (*derby* and *xml.transform*) failed to be compiled because adaptive compilation is disabled. The original *SPECjbb2005* runs for a fixed amount of time and is not suitable for evaluating performance slowdown. To ensure a fixed workload, we evaluate a variant one called *Pseudojbb2005*, which runs for a fixed number of transactions (100000).

5.1 Slowdown in Record Phase

Figure 7 depicts the performance slowdown of record phase in ORDER. All benchmarks are run with 16 threads to evaluate the performance slowdown. To show the effect of eliminating unnecessary timeline recording, performance before and after this optimization are both presented (before-opt vs. after-opt). In the raw recording system, although most of the applications have performance slowdown of less than 8 times and overhead of some benchmarks is even lower than 100%, the slowdown rises up to 82x in an extreme case (*compress*). Compared to the raw system, performance after eliminating unnecessary timeline recording is much better. As shown in Figure 7, after optimization, the record overhead of ORDER is less than 8 times in all cases, even in *compress*. ORDER incurs an average of 108% overhead compared to the original execution run. Actually, the optimized version of ORDER introduces less than 100% overhead for most of the benchmarks, which means that it may be efficiently used for many applications. If not specifically mentioned, all results below are collected under optimized ORDER system.

Moreover, we evaluated three more configurations of ORDER to investigate the source of overhead in ORDER: raw Apache Harmony without adaptive compilation (wo_adaptive), ORDER without recording timeline (wo_timeline), and ORDER without swapping timeline log to disk, i.e., the timeline records are stored only in memory (wo_disk). As shown in Figure 7, the performance of most applications (except *JRuby*) after disabling adaptive compilation is very close to the original JVM. On average, there is a 5.8% performance difference, which means performance impact of adaptive compilation is insignificant in most cases. Besides, the performance of ORDER is similar whether disabling disk operations for timeline logs or not. These two configurations introduce a slowdown of about 104% and 108% correspondingly. This shows that overhead of disk operation is also small. Further, when disabling timeline recording as a whole, ORDER introduces only 16% overhead on average. This confirms that the major performance overhead of ORDER comes from tracing timeline in memory.

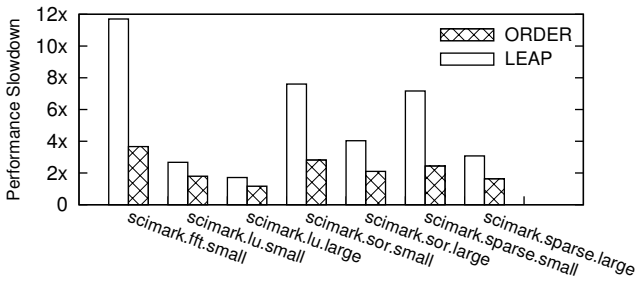


Figure 8: Performance slowdown of record phase in ORDER and LEAP. 16 threads are used to evaluate the performance slowdown.

Figure 8 depicts the performance of ORDER compared to LEAP[15]. Because the static instrumentation approach does not support reflection, LEAP cannot properly instrument Java code of original SPECjvm2008 or SPECjbb2005. To evaluate the performance of LEAP, we manually replace the reflection mechanism in *scimark*, *mpegaudio*, and *compress* with direct method invocation. We found that when recording *mpegaudio*, *compress*, *scimark.monte_carlo* or *scimark.fft.large*, LEAP either throws an OutOfMemoryError or does not finish in two hours. Results of JRuby are also not presented here because LEAP throws NULLPointerException in static instrumentation phase. The performance result just serves as a reference because LEAP records non-determinism in neither library code nor Java runtime. As shown in Figure 8, although ORDER records more non-deterministic events than LEAP, ORDER is still 1.4x to 3.2x faster than LEAP in the evaluated benchmarks.

By reducing the strength of synchronization, ORDER notably improves the scalability of recording interleaved object accesses in Java applications. Figure 9 shows the performance slowdown of ORDER when the number of threads varies from 1 to 16. Overall, ORDER scales well.

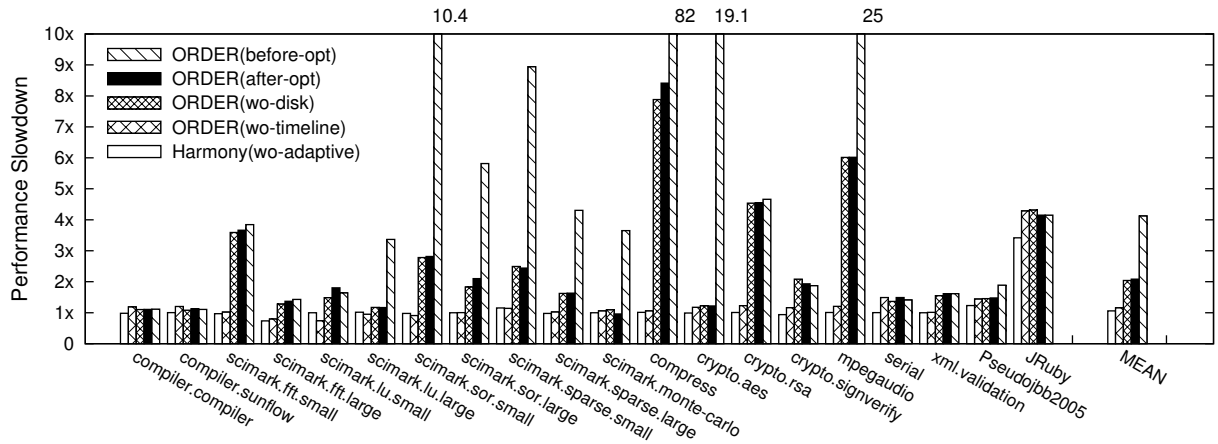


Figure 7: Performance slowdown of record phase in ORDER. 16 threads are used to evaluate the performance slowdown.

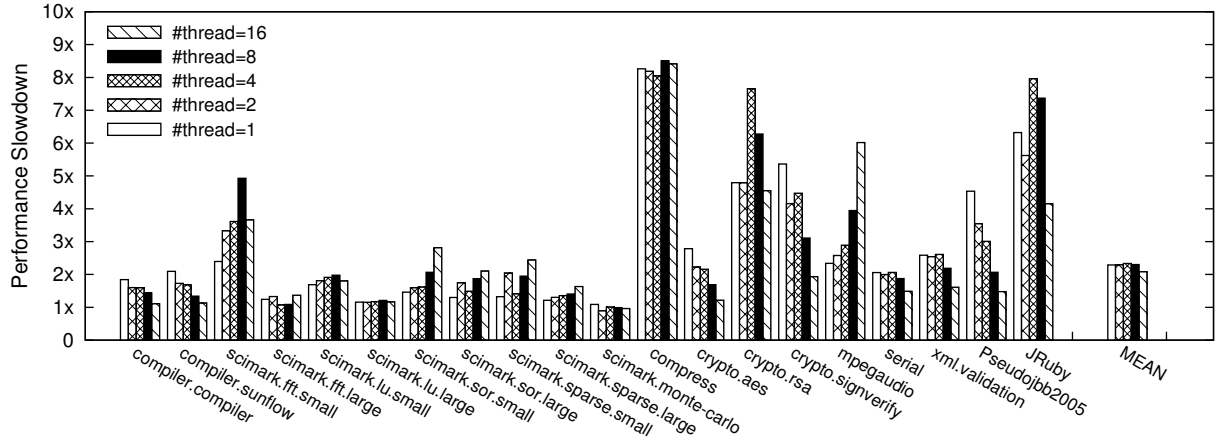


Figure 9: Performance slowdown of record phase when the number of threads varies from 1 to 16.

Most of the cases have similar performance slowdown while the number of threads increases. With the number of threads increasing, only one case in SPECjvm2008 (*mpegaudio*) has an obvious increase of performance slowdown. The increased number of object access interleavings degrades the performance in this case. Meanwhile, we observed that there are still many assigned-once objects recorded in this case, which is caused by the conservativeness of assigned-once analysis, and they can be further eliminated by improving the precision of analysis algorithm. For some applications such as *compiler* and *crpto.aes*, when we increase the number of threads, the recording overhead even decreases. Despite the reduced contention of ORDER, this anomaly is also affected by the following two reasons: 1) As discussed in Yi et al. [34], some benchmarks themselves are not scalable, like *compiler*; 2) Instrumentation of ORDER increases the complexity of intermediate representation, thus introduces additional overhead to analysis and optimizations in Harmony; such overhead is amortized when the number of threads increases.

5.2 Slowdown in Replay Phase

Similarly, ORDER uses dynamic instrumentation in Harmony to implement the replay phase. Thus, instrumentation of replay phase causes similar performance slowdown to record phase. Besides, blocking threads to ensure correct timeline will introduce additional overhead. Figure 10 depicts the replay slowdown of ORDER. For most of the selected benchmarks, the performance of replay phase is scalable from 1 to 16 threads, with four exceptions(*xml.validation*, *serial*, *mpegaudio*, and *Pseudojbb2005*). In these four applications, with the number of threads increasing, performance slowdown also increases because Java threads are frequently blocked. Currently the prototype of ORDER uses a naive implementation of thread scheduler, which can be further enhanced to speedup replay phase performance.

5.3 Log Size

Besides the performance overhead, many state-of-the-art deterministic replay systems also suffer from large space overhead. To record interleaved data accesses on

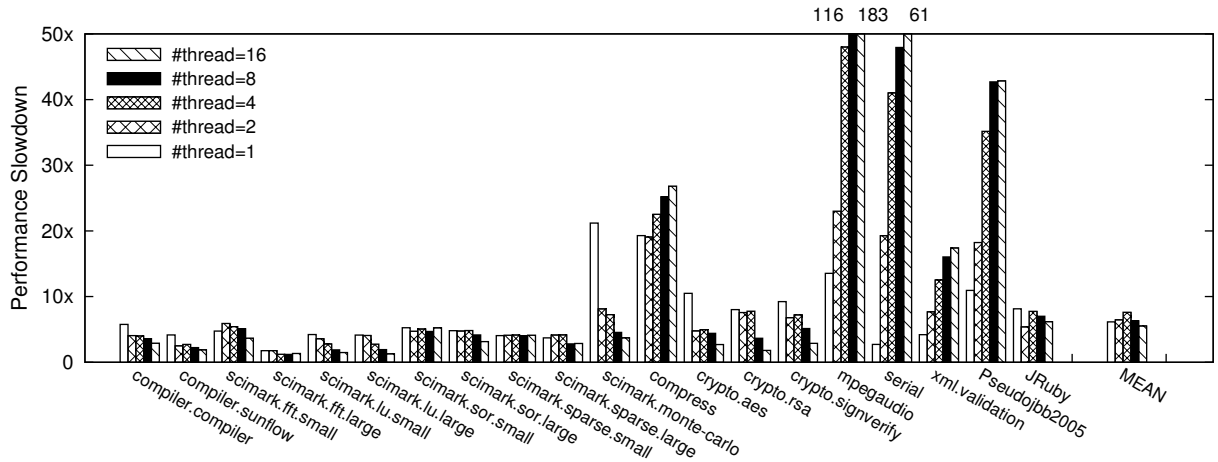


Figure 10: Performance slowdown of replay phase when the number of threads varies from 1 to 16.

multi-processor architecture, deterministic replay systems usually need to record tens of Gigabytes logs per Hour(g/h) [12, 15]. Although the disk capacity today is large enough to store log files, it is hard to share such large data on network. If we want to report a concurrency bug to the corresponding community, uploading a log file with tens of gigabytes is obviously not attractive.

Table 2 shows that the log size of ORDER is very small. In most cases, it generates a log file less than 100 Megabytes per Hour(m/h), which is considerably smaller than those reported in other deterministic replay systems[12, 15]. Only two cases (*serial* and *Pseudojbb2005*) generate log files that are greater than 1 Gigabytes per Hour. We notice that most of the logged interleavings of *serial* relate to contention for a single global buffer in the original application, which is introduced by a producer-consumer scenario. Most of the logged interleavings in *Pseudojbb2005* are caused by the false sharing between different static fields in the same class, which occurs because prototype of ORDER does not distinguish accesses to the same class object. However, such a log size is still much smaller than those reported in previous literatures [12, 15].

Case	Log Size (timeline)	Log Size (others)
compiler.compiler	88(m/h)	35(m/h)
compiler.sunflow	61(m/h)	58(m/h)
scimark.fft.small	0.60(m/h)	10(m/h)
scimark.fft.large	0.47(m/h)	7(m/h)
scimark.lu.small	0.37(m/h)	6(m/h)
scimark.lu.large	0.35(m/h)	5(m/h)
scimark.sor.small	2(m/h)	40(m/h)
scimark.sor.large	0.68(m/h)	11(m/h)
scimark.sparse.small	2(m/h)	36(m/h)
scimark.sparse.large	0.56(m/h)	10(m/h)
scimark.monte-carlo	0.013(m/h)	0.22(m/h)
compress	4(m/h)	44(m/h)
crypto.aes	1.4(m/h)	9(m/h)
crypto.rsa	26(m/h)	6(m/h)
crypto.signverify	10(m/h)	8(m/h)
mpegaudio	511(m/h)	2(m/h)
serial	1553(m/h)	121(m/h)
xml.validation	632(m/h)	31(m/h)
Pseudojbb2005	1085(m/h)	550(m/h)
JRuby	0.8(m/h)	170(m/h)

Table 2: Log size of ORDER, in 16-thread execution.

5.4 Concurrency Bug Reproducibility

To confirm the reproducibility of ORDER on concurrency bugs, we reproduce six real-world concurrency bugs from open source projects with ORDER. The characteristics of these concurrency bugs are listed in Table 3. These cases cover three major categories of concurrency bugs reported by Lu et al. [18]. By replaying the recorded logs, the buggy executions are successfully reproduced in replay phase. Among the bugs, JRuby-2483 is caused by using thread unsafe library code, which fails to be reproduced in a static instrumentation approach [15].

6 Related Work

State-of-the-art deterministic replay for Java: State-of-the-art deterministic replay systems for Java applications use the strategy called "logical thread scheduling" to record multi-threaded Java execution [28, 9, 30]. As mentioned in Dejavu [9], "logical thread scheduling" *is based on a global clock (i.e., time stamp) for the entire application*. This strategy works efficiently in uniprocessor platforms. However, global clock among cores needs to be synchronized frequently, which imposes contention to a single global lock. There are currently no scalable deterministic replay systems based-on such an approach for multi-processor platforms.

JaRec [13] assumes that Java applications are data-

Bug ID	Category	Bug description
JRuby-931	atomic violation	Non-atomic traversing of container triggers ConcurrentModification-Exception.
JRuby-1382	atomic violation	Non-atomic read from memory cache causes system crash.
JRuby-2483	atomic violation	Concurrency bug caused by using thread unsafe library code.
JRuby-879 JRuby-2380	order violation	Listing threads before thread is registered causes non-deterministic result.
JRuby-2545	deadlock	Lock on the same object twice causes deadlock.

Table 3: Real-world concurrency bugs reproduced by ORDER. Each of them comes from open source communities and causes real-world buggy execution.

race free programs and records only the lock acquisition order, which cannot be used to reproduce concurrency bugs caused by data races. LEAP [15] records non-determinism introduced by data accesses through static recompilation and instrumentation, which cannot cover external code, such as libraries or class files dynamically loaded during runtime. Thus, it cannot reproduce concurrency bugs caused by these missing parts. None of the existing deterministic replay systems can reproduce bugs caused by non-determinism inside JVM. Furthermore, LEAP does not distinguish instances of the same class, and the false sharing between different objects may lead to large performance overhead when a class is massively instantiated.

There are also several proposed approaches to improve efficiency and scalability of deterministic replaying native code written in C/C++. They can be grouped into two sets according to how they record non-deterministic data accesses:

Software deterministic replay for native code: Uniprocessor deterministic replay systems [5, 33] record interrupt boundaries and input payloads, which are proven useful in bug diagnosis and intrusion detection. However, state-of-the-art systems use multiple CPUs with shared memory data access. Such an additional source of non-determinism makes efficient recording difficult for software.

Several approaches are proposed to reduce the synchronization overhead and performance slowdown introduced by memory race recording. Transitive Reduction [23] is proposed to reduce the log size by applying transitivity-based log reduction to log files generated by deterministic replay system. It can also reduce the synchronization overhead in the replay phase. However, such an approach still needs to use global clock and cannot reduce synchronization overhead in record phase.

SMP-Revirt [12] modifies the page protection mechanism for recording non-deterministic data access events. By using page as the granularity to track dependencies, SMP-Revirt achieves a low performance overhead in architectures with 1 to 2 cores. However, because they record a very large granularity of data sharing, its performance drops significantly when number of cores is increased to 4. In order to mitigate thrashing caused by frequent transfers of page ownership, SCRIBE [16] defines a minimal ownership retention interval and disallows ownership transitions until the interval expires. Although it notably relieves the contention among threads, the extended interval of page ownership makes it difficult to capture atomic violation bugs in record run.

PRES and ODR [26, 4] record partial information in record phase and use an offline reproducer to infer the race occurred in the record phase. Because only a part of the execution information is recorded, they can achieve a low performance slowdown. However, the reproducibility depends heavily on how much information is recorded, and the proper scheme to record information is hard to decide. Although recording less information can reduce performance overhead, the recorded execution may not be reproducible.

Hardware-assisted deterministic replay: Since software based deterministic replay systems usually impose large performance overhead, hardware-assisted deterministic replay systems [6, 14, 17, 19, 20, 21, 22, 32] are proposed to modify hardware components for recording data access conflicts efficiently. Many such systems apply optimizations to further reduce record overhead and log size. However, they impose non-trivial hardware complexity and there are still no commercially available processors built with these features.

7 Conclusion and Future work

This paper presented ORDER, the first object-centric deterministic replay system for concurrent Java applications on multicore. ORDER recorded interleaved data accesses in Java applications by tracking how each thread accesses each object and enforced such a constraint during replay. By dynamically instrumenting Java code in the compilation pipeline, ORDER naturally covered non-determinism in dynamically loaded classes and libraries. Evaluation results showed that ORDER achieved good performance and scalability for a range of benchmarks, which notably outperformed LEAP, a state-of-the-art deterministic replay system for Java. Bug reproducibility study further showed that ORDER successfully reproduced several real-world concurrency bugs.

While ORDER has demonstrated the efficiency and

effectiveness of recording and replay concurrent Java programs, there are still plenty of optimization spaces, which will be our future work. First, timeline filter of unnecessary dependencies is currently applied offline, and applying it online can reduce space overhead and eliminate unnecessary disk operations in the recording phase. Second, we plan to combine ORDER with techniques that cover the non-determinism in adaptive optimization [24] to enable adaptive optimization for JVM. Finally, we plan to combine ORDER with an object-level checkpointing mechanism to further reduce log size, and extend ORDER with some selective tracking mechanisms to focus on only interested objects, to further reduce performance overhead.

Acknowledgments

We thank our shepherd Keith Adams and the anonymous reviewers for their insightful comments. This work was funded by China National Natural Science Foundation under grant numbered 61003002 and 90818015, a grant from the Science and Technology Commission of Shanghai Municipality numbered 10511500100, IBM X10 Innovation Faculty Award, a research grant from Intel as well as a joint program between China Ministry of Education and Intel numbered MOE-INTEL-09-04, Fundamental Research Funds for the Central Universities in China and Shanghai Leading Academic Discipline Project (Project Number: B114).

References

[1] Apache harmony. http://harmony.apache.org.

[2] Fault tolerance in vmware. http://www.vmware.com/files/pdf/perf-vsphere-fault_tolerance.pdf.

[3] Optimizations in harmony. http://wiki.apache.org/harmony/Jitrino_OPT.

[4] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *Proc. SOSP*, pages 193–206, 2009.

[5] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *Proc. VEE*, pages 154–163, 2006.

[6] Y. Chen, W. Hu, T. Chen, and R. Wu. Lreplay: a pending period based deterministic replay scheme. In *Proc. ISCA*, pages 187–197, 2010.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proc. OOPSLA*, pages 1–19, 1999.

[8] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. PLDI*, pages 258–269, 2002.

[9] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proc. SIGMETRICS symposium on Parallel and distributed tools(SPDT)*, pages 48–59, 1998.

[10] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. USENIX ATC*, pages 1–14, 2008.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proc. OSDI*, pages 211–224.

[12] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution replay of multiprocessor virtual machines. In *Proc. VEE*, pages 121–130, 2008.

[13] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. Jarec: a portable record/replay environment for multi-threaded java applications. *Softw. Pract. Exper.*, 34(6):523–547, 2004.

[14] D. R. Hower and M. D. Hill. Rerun: Exploiting episodes for lightweight memory race recording. In *Proc. ISCA*, pages 265–276, 2008.

[15] J. Huang, P. Liu, and C. Zhang. Leap: The lightweight deterministic multi-processor replay of concurrent java programs. In *Proc. FSE*, 2010.

[16] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In *Proc. SIGMETRICS*, pages 155–166, 2010.

[17] D. Lee, M. Said, S. Narayanasamy, Z. Yang, and C. Pereira. Offline symbolic analysis for multi-processor execution replay. In *Proc. MICRO*, pages 564–575, 2009.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, pages 329–339, 2008.

[19] P. Montesinos, L. Ceze, and J. Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proc. ISCA*, pages 289–300, 2008.

[20] P. Montesinos, M. Hicks, S. T. King, and J. Torrellas. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proc. ASPLOS*, pages 73–84, 2009.

[21] S. Narayanasamy, C. Pereira, and B. Calder. Recording shared memory dependencies using strata. In *Proc. ASPLOS*, pages 229–240, 2006.

[22] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proc. ISCA*, pages 284–295, 2005.

[23] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. ACM/ONR workshop on Parallel and distributed debugging(PADD)*, pages 1–11, 1993.

[24] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: improving debuggability of a just-in-time compiler. In *Proc. OOPSLA*, pages 241–252, 2006.

[25] S. Park, S. Lu, and Y. Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proc. ASPLOS*, pages 25–36, 2009.

[26] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, pages 177–192, 2009.

[27] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proc. CGO*, pages 2–11, 2010.

[28] K. Ravi, S. Harini, and C. Jong-Deok. Deterministic replay of distributed java applications. In *Proc. IPDPS*, page 219, 2000.

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. *SIGOPS Oper. Syst. Rev.*, 31(5):27–37, 1997.

[30] V. Schuppan, M. Baur, and A. Biere. Jvm independent replay in java. *Elsevier Electron. Notes Theor. Comput. Sci.*, 113:85–104, 2005.

[31] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proc. conference of the Centre for Advanced Studies on Collaborative research(CASCON)*, pages 13–, 1999.

[32] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. ISCA*, pages 122–135, 2003.

[33] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proc. Annual Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2007.

[34] Y. Zhao, J. Shi, K. Zheng, H. Wang, H. Lin, and L. Shao. Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In *Proc. OOPSLA*, pages 361–376, 2009.

Enabling Security in Cloud Storage SLAs with CloudProof

Raluca Ada Popa
M.I.T.

Jacob R. Lorch, David Molnar, Helen J. Wang, and Li Zhuang
Microsoft Research

Abstract

Several cloud storage systems exist today, but none of them provide security guarantees in their Service Level Agreements (SLAs). This lack of security support has been a major hurdle for the adoption of cloud services, especially for enterprises and cautious consumers. To fix this issue, we present *CloudProof*, a secure storage system specifically designed for the cloud. In CloudProof, customers can not only *detect* violations of integrity, write-serializability, and freshness, they can also *prove* the occurrence of these violations to a third party. This proof-based system is critical to enabling security guarantees in SLAs, wherein clients pay for a desired level of security and are assured they will receive a certain compensation in the event of cloud misbehavior. Furthermore, since CloudProof aims to scale to the size of large enterprises, we delegate as much work as possible to the cloud and use cryptographic tools to allow customers to detect and prove cloud misbehavior. Our evaluation of CloudProof indicates that its security mechanisms have a reasonable cost: they incur a latency overhead of only $\sim$15% on reads and writes, and reduce throughput by around 10%. We also achieve highly scalable access control, with membership management (addition and removal of members' permissions) for a large proprietary software with more than 5000 developers taking only a few seconds per month.

1 Introduction

Storing important data with cloud storage providers comes with serious security risks. The cloud can leak confidential data, modify the data, or return inconsistent data to different users. This may happen due to bugs, crashes, operator errors, or misconfigurations. Furthermore, malicious security breaches can be much harder to detect or more damaging than accidental ones: external adversaries may penetrate the cloud storage provider, or employees of the service provider may commit an insider attack. These concerns have prevented security-conscious enterprises and consumers from using the cloud despite its benefits [16].

These concerns are not merely academic. In June 2008, Amazon started receiving public reports that data on its popular Simple Storage Service (S3) had been corrupted due to an internal failure; files no longer matched customers' hashes [11]. One day later, Amazon confirmed the failure, and cited a faulty load balancer that had corrupted single bytes in S3 responses "intermittently, under load." Another example of data security violation in the cloud occurred when Google Docs had an access-control bug that allowed inadvertent sharing of documents with unauthorized readers [28]. Even worse, LinkUp (MediaMax), a cloud storage provider, went out of business after losing 45% of client data because of administrator error.

None of today's cloud storage services—Amazon's S3, Google's BigTable, HP, Microsoft's Azure, Nirvanix CloudNAS, or others—provide security guarantees in their Service Level Agreements (SLAs). For example, S3's SLA [1] and Azure's SLA [23] only guarantee availability: if availability falls below 99.9%, clients are reimbursed a contractual sum of money. As cloud storage moves towards a commodity business, security will be a key way for providers to differentiate themselves. In this paper, we tackle the problem of designing a cloud storage system that makes it possible to detect violations of security properties, which in turn enables meaningful security SLAs.

The cloud security setting is different from the setting of previous secure storage or file systems research. The first difference is that there is a financial contract between clients and the cloud provider: clients pay for service in exchange for certain guarantees and the cloud is a liable entity. In most previous work [3, 4, 10, 20], the server was some group of untrusted remote machines that could not guarantee any service. The second difference is that scalability is more important, as it is one of the primary promises of the cloud. Enterprises are important customers for the cloud; they have many employees requiring highly scalable access control and have large amounts of data.

We identify four desirable security properties of cloud storage: confidentiality, integrity, write-serializability, and read freshness (denoted by C, I, W, F). If a customer has such security guarantees, his data is confiden-

tial, cannot be modified by any unauthorized party, is consistent among updates made by authorized users, and is fresh as of the last update.

We design, build, implement, and evaluate CloudProof, a secure and practical storage system specifically designed for the cloud setting. Our first novelty is the idea and the mechanism of enabling customers *to prove to third parties when the cloud violates the IWF properties*. (Confidentiality is not included because customers can provide it to themselves by encrypting the data they store on the cloud.) This enabling of proofs is in addition to detecting the violations and is not present in previous work. It includes the fact that *the cloud can disprove false accusations made by clients*; that is, in CloudProof, clients cannot frame the cloud. We believe that such proofs are *key* to enabling security in SLAs with respect to these three properties. Customers and cloud can now establish a financial contract by which clients pay a certain sum of money for the level of security desired; customers have assurance that the cloud will pay back an agreed-upon compensation in case their data security is forfeited because they can prove this violation. Without such proofs, the cloud can claim a smaller amount of damage to protect itself against significant financial loss and clients can falsely accuse the cloud. These proofs are based on *attestations*, which are signed messages that bind the clients to the requests they make and the cloud to a certain state of the data. For every request, clients and cloud exchange attestations. These attestations will be used in a lightweight auditing protocol to verify the cloud's behavior.

The second novelty is CloudProof, the system as a whole, in which we put engineering effort to maintain cloud scalability while detecting and proving violations to all three IWF properties and providing access control. Previous work did not provide detection for both write-serializability and freshness at the same time. In addition, most related work has not been designed with cloud scalability in mind and we argue that they are not readily extendable to provide it. Our design principle is to *offload as much of the work as possible to the cloud, but verify it*. Therefore, access control, key distribution, read, write, file creation, and ensuring the aforementioned security properties are delegated to the cloud to the extent possible. To enable this delegation, we employ cryptographic tools from the literature: to achieve scalable access control, we use in a novel way key rolling and broadcast encryption. We also have a novel way to group data by access control list into "block families" that allows us to easily handle changes in access control.

CloudProof targets most applications that could benefit from the cloud: large departmental or enterprise file systems, source code repositories, or even small, personal file systems. These tend to be applications that can tolerate larger client-cloud latency (which is an inherent result of the different geographic locations of various clients/organizations with respect to the cloud). Yet, a surprising number of applications benefit from the cloud. For example, the Dropbox service uses S3 storage to provide backup and shared folders to over three million users. The SmugMug photo hosting service has used S3 since April 2006 to hold photos, adding ten terabytes of data each month without needing to invest in dedicated infrastructure. AF83 and Indy500.com use S3 to hold static web page content.

Any security solution for cloud storage must have a limited performance impact. We have prototyped CloudProof on Windows Azure [22]. In Section 9 we report experiments that measure the latency and throughput added by CloudProof compared to the storage system without any security. In microbenchmarks, for providing all four of our properties, we add $\sim$0.07s of overhead ($\sim$15%) to small block reads or writes, and achieve only $\sim$10% throughput reduction and 15% latency overhead for macrobenchmarks. One can audit the activity of a large company during a month in 4 min and perform membership changes (adding and revoking member permissions) for a source code repository of a very large proprietary software that involved more than 5000 developers in a few seconds per month. Overall, our evaluations show that we achieve our security properties at a reasonable cost.

2 Setting

CloudProof can be built on top of conventional cloud storage services like Amazon S3 or Azure Blob Storage. The storage takes the form of key-value pairs accessed through a `get` and `put` interface: the keys are block IDs and the values are the contents of the blocks. Blocks can have any size and can vary in size.

There are three parties involved in CloudProof:

1. *(Data) owner:* the entity who purchases the cloud storage service. A data owner might be an enterprise with business data or a home user with personal data.
2. *Cloud:* the cloud storage provider.
3. *(Data) users:* users who are given either read or write access to data on the cloud. A user might be an employee of an enterprise or family members and friends of a home user.

The data owner is the only one allowed to give access permissions to users. The access types are read and read/write. Each block has an access control list (ACL), which is a list of users and their accesses to the block. (One can easily implement a group interface by organizing users in groups and adding groups to ACLs.) When talking about reading or modifying a block, a *legitimate* user is a user who has the required access permissions to the block. We assume that the data owner and the cloud have well-known public keys, as is the case with existing

providers like Amazon S3.

2.1 Threat Model

The cloud is *entirely untrusted*. It may return arbitrary data for any request from the owner or any user. Furthermore, the cloud may not honor the access control lists created by the owner and send values to a user not on the corresponding access control list. A user is trusted with the data he is given access to. However, he may attempt to subvert limits on his permission to access data, possibly in collusion with the cloud. An owner is trusted with accessing the data because it belongs to him. However, the users and the owner may attempt to falsely accuse the cloud of violating one of our security properties.

We make standard cryptographic assumptions for the tools we use: existential unforgeability under chosen message attack for public-key signature schemes, collision-resistance and one-way function property for hash functions, and semantic security for symmetric encryption schemes.

2.2 Goals

Let us first define the security properties CloudProof provides. *Confidentiality (C)* holds when the cloud or any illegitimate user cannot identify the contents of any blocks stored on the cloud. *Integrity (I)* holds when each read returns the content put by a legitimate user. For example, the cloud cannot replace some data with junk. *Write-serializability (W)* holds when each user committing an update is aware of the latest committed update to the same block. W implies that there is a total order on the writes to the same block. *Freshness (F)* holds if reads return the data from *the latest committed write.* Note that we cannot guarantee that each block retrieved was the most recently received by the cloud, because, upon two parallel writes to the block, the cloud can pretend to have received them in a different order or network delays can arbitrarily reorder such requests. Instead, we aim to guarantee that the last committed write (for which the cloud acknowledged receipt to the client, as we will see) will be visible during read. A violation to the security of a user is when the IWF properties do not hold.

CloudProof has the following four goals.

Goal 1: Users should detect the cloud's violations of integrity, freshness, and write-serializability. Users should provide confidentiality to themselves by encrypting the data they store on the cloud.

Goal 2: Users should be able to *prove* cloud violations whenever they happen. Any proof system has two requirements: (1) the user can convince a third party of any true cloud violation; and (2) the user cannot convince a third party when his accusation of violation is false.

Goal 3: CloudProof should provide read and write access control in a scalable (available) way. Since we are targeting enterprise sizes, there may be hundreds of thousands of users, many groups, and terabytes of data. We want to remove data owners from the data access path as much as possible for performance reasons. Owners should be able to rely (in a verifiable way) on the cloud for key distribution and access control, which is a highly challenging task.

Goal 4: CloudProof should maintain the performance, scalability, and availability of cloud services despite adding security. The overhead should be acceptable compared to the cloud service without security, and concurrency should be maintained. The system should scale to large amounts of data, many users per group, and many groups, since this is demanded by large enterprise data owners.

In the remainder of this paper, we show how CloudProof achieves these goals.

3 System Overview

In this section, we present an overview of our system; we will elaborate on each component in later sections.

CloudProof's interface consists of get(BlockID blockID) and put(BlockID blockID, byte[] content). BlockID is a flat identifier that refers to a block on the cloud. The get command reads content of a block, while the put command writes in the block identified by blockID. Creation of a block is performed using a put with a new blockID and deletion is performed by sending a put for an empty file. CloudProof works with any cloud storage that exports the following key/value store interface and can sign messages verifiable by other parties using some known public key.

The design principle of CloudProof is to *offload as much work as possible to the cloud, but be able to verify it*. The cloud processes reads and writes, maintains data integrity, freshness, and write-serializability, performs key distribution (protected by encryption), and controls write accesses. Users and the owner verify the cloud performed these operations correctly.

We put considerable engineering effort into keeping the system scalable. The data owner only performs group membership changes and audits the cloud. Both tasks are offline actions and lightweight, as we will see in Section 9. Thus, the data owner is not actively present in any data access path (i.e. put or get), ensuring the service is scalable and available even when the data owner is not. Moreover, access control is delegated to the cloud or distributed. As part of access control, key distribution is delegated in a novel way using broadcast encryption and key rolling to the cloud: the data owner only needs to change one data block when a user is revoked rather than all the blocks the user had access to. Most data reencryption is distributed to the clients for scalability. Importantly, we strived to keep accesses to different blocks running in parallel and avoided the typical serialization

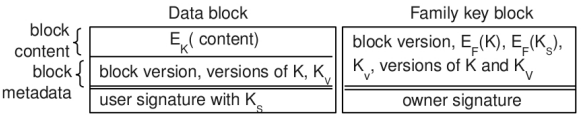


Figure 1: Layout of block formats in a family and of the key block. K is the read access key, K_S is the signing key, and K_V is the verification key. $E_F()$ means broadcast encryption to the authorized users in the ACL of a certain block family. The block version is an integer for each block that is incremented by one with every update to the block committed on the cloud.

so often encountered in security protocols (e.g., computations of Merkle trees over multiple blocks upon data write).

The key mechanism we use is the exchange of *attestations* between the owner, users, and the cloud. When users access cloud storage through the get or put interface, each request and response is associated with an attestation. Attestations keep users and the cloud accountable and allow for later proofs of misbehavior, as we discuss in Section 5.

We divide time into *epochs*, which are time periods at the end of which data owners perform auditing. At the end of each epoch, owners also perform membership changes such as addition or removal of members. (Owners can also change membership during an epoch and clients will have to check if any keys have changed due to revocations.) Each epoch has a corresponding *epoch number* that increases with every epoch. If the system uses fixed-length epochs, clients can easily derive the current epoch identifier independently from the current time.

Briefly, CloudProof detects and proves IWF violations as follows. Clients check data integrity based on attestations they get from the cloud, as described in Section 6. The owner checks W and F during the *auditing* process. For auditing efficiency, each block has a certain probability of being audited in an epoch. During the epoch, users send the attestations they receive from the cloud to the owner. If they are disconnected from the owner, they can send these attestations any time before the epoch's end; if they are malicious or fail to send them for any reason, there is no guarantee that their gets returned correct data or their puts took effect. The owner uses these attestations to detect any violations of WF and construct a proof that convinces a third party whenever the cloud misbehaved, as we will discuss in Section 7.

4 Access Control

In this section, we focus only on access control; as such, we consider that the cloud does not modify the data placed by authorized users and does not provide stale or inconsistent views to users: we provide mechanisms to enforce these properties separately in Sections 6 and 7.

If the owner adds or removes a user from the ACL of a block, then that user gains or loses access to that block.

We introduce the term **block family** to describe the set of all blocks that have the same ACL. If the owner of a block changes its ACL, the block will switch to a different family. Since all blocks in a family have identical ACLs, when we need a key to enforce access control we can use the same key for every block in a family.

As mentioned, the cloud is not trusted with access control. At the same time, having the owner perform access control checks for every user get or put would be costly and unscalable. We thus follow our design principle of verifiably offloading as much work as possible to the cloud.

Read Access Control. To prevent unauthorized reads, all the data stored on the cloud is encrypted with a secure block or stream cipher, e.g., AES in counter mode. We denote the secret key of the cipher as the *read/get access key*. Clients with read access will have the key for decryption as described in Section 4.1 and thus will be able to access the data. Blocks in the same block family will use the same read access key.

Write Access Control. We achieve write access control with a public key signature scheme, as follows. For each block family, we have a public *verification key* and a private *signing key*. The verification key is known to everyone, including the cloud, but the signing key is known only to users granted write access by the ACL. Each time a user modifies a block, he computes its *integrity signature*, a signature over the hash of the updated block using the signing key. He sends this signature along with his write request, and the cloud stores it along with the block. The cloud provides it to readers so they can verify the integrity of the block.

Since the verification key is known to the cloud, it can perform write access control, as follows. Whenever a user attempts to update a block, the cloud verifies the signature and only allows the update if the signature is valid. Note that, if the cloud mistakenly allows a write without a valid signature, this failure will be detected by future data users reading the block. The mechanism of attestations, described later, will allow those users to prove this cloud misbehavior.

4.1 Key distribution

Our key distribution mechanism ensures that users can acquire the keys they need to access the blocks they are authorized to access. To offload as much work as possible to the cloud, the cloud performs this key distribution verifiably. We achieve this goal by employing in a novel way broadcast encryption and key rolling. **Broadcast encryption**([5, 12]) allows a broadcaster to encrypt a message to an arbitrary subset of a group of users. Only the users in the subset can decrypt the message. Encrypting creates a ciphertext of size $O(\sqrt{\text{total no. of users in the group}})$. **Key rotation** [18] is a scheme in which a sequence of keys can be produced

from an initial key and a secret master key. Only the owner of the secret master key can produce the next key in the sequence, but any user knowing a key in the sequence can produce all earlier versions of the key (forward secrecy).

For each block family, the owner places one block on the cloud containing the key information for that family. This block is called the *family key block*. Only the data owner is allowed to modify the family key block. Recall that all blocks in a family share the same ACL, so each key block corresponds to a particular ACL that all blocks in its family share. Figure 1 illustrates the layout of blocks in a family and of the key block. The purposes of the various terms in the figure are explained in the rest of this section.

Using broadcast encryption, the data owner encrypts the read access key so that only users and groups in the ACL's read set can decrypt the key. This way, only those users and groups are able to decrypt the blocks in the corresponding family. The data owner also uses broadcast encryption to encrypt the signing key so that only users and groups in the ACL's write set can decrypt the key. This way, only those users and groups can generate update signatures for blocks in the corresponding family.

We do not try to prevent a data user giving his read access key for a block family to a data user who is not authorized for that block family. The reason is that authorized data users can simply read the information directly and give it to others. Solving this problem, e.g., with digital rights management, is beyond the scope of this paper.

4.2 Granting/revoking access

The owner may want to revoke the access of some users by removing them from certain groups or from individual block ACLs. When the owner revokes access of a user, he should make sure that the data user cannot access his data any more. For this goal, there are two options, each appropriate for different circumstances. *Immediate revocation* that the revoked user should not have access to any piece of data from the moment of the revocation. *Lazy revocation* means that the revoked user will not have access to any data blocks that have been updated after his revocation. The concept of lazy revocation is in [13] and is not new to us.

When a block's ACL changes, that block must undergo immediate revocation. That is, the block must switch to a new family's key block, the one corresponding to its new ACL, and the block needs to be immediately re-encrypted with that new key block.

In contrast, when a group's membership changes, then all blocks with ACLs that include that group must undergo revocation. Using immediate revocation in this case would be too expensive, as it would involve immediately re-encrypting all the blocks in one or more block families, a potentially immense amount of data. Furthermore, such an approach may be futile because a malicious revoked data user could have copied all the blocks for which he had access. Instead, we use lazy revocation, as follows.

Using key rotation, the owner rolls the keys forward to a new version for each of the families corresponding to the affected ACLs. However, the blocks with those ACLs do not need to be re-encrypted right away; they can be lazily re-encrypted. The owner only needs to update the family key blocks with the new key information. This means the work the owner has to do upon a membership change is *independent of the number of files in the block family*. Broadcast encryption has complexity $O(\sqrt{\text{no. of members in ACL}})$, which we expect to be manageable in practice, as we will show in Section 9.

When a user accesses a block, he checks whether the version of the read access key in the family key block is larger than the version of the key with which the current block was encrypted. If so, the data user re-encrypts the block with the new key. Re-encrypting with a different key does not incur any overhead since all writes require a re-encryption. Therefore, the burden of the revocation is pushed to users, but without them incurring any additional re-encryption overhead.

We can see that our division of storage into block families makes revocation easier. If, in contrast, there were multiple Unix-like groups for a block, one would need to store encryptions of the signature and block encryption key for every group with each block. Besides the storage overhead, this would require many public key operations whenever a member left a group because the key would need to be changed for every regular group. This process would be slower and more complex.

5 Attestations

In this section, we describe the structure and exchange of the attestations[1]. The attestations are key components that allow the clients to prove cloud misbehavior and the cloud to defend himself against false accusations.

Every time the client performs a get, the cloud will give the client a *cloud get attestation*. Every time a client performs a put, the client will give the cloud a *client put attestation* and the cloud will return a *cloud put attestation*. Intuitively, the role of the attestations is to *attest* to the behavior of each party. When the client performs a get, the cloud attaches to the response an attestation; the attestation is similar to the cloud saying "I certify that I am giving you the right data". When the client performs a put, he must provide a client put attestation which intuitively says "I am asking you to overwrite the existing

[1] CloudProof's attestations should not be confused with attestations from trusted computing, which are different mechanisms.

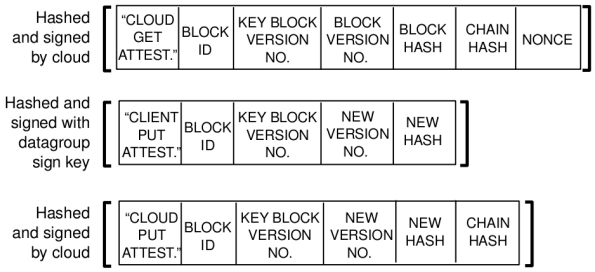


Figure 2: The structure of the attestations. The elements listed in each attestation are concatenated, a hash is computed over the concatenation and the result is signed as shown in the figure. The first field indicates the type of the attestation. The hash in client attestations is a hash over the block content and metadata, while the hash in the cloud attestation includes the integrity signature as well; the new hash is the value of this hash after an update. The nonce is a random value given by the client.

data with this content". The cloud must answer with a cloud put attestation saying "The new content is committed on the cloud".

5.1 Attestation Structure

Each attestation consists of concatenating some data fields, and a signed hash of these data fields. Since we are using signatures as proofs, it is important that they be non-repudiable (e.g. *unique signatures* [21]). Figure 2 illustrates the structure of the attestations. The *block version number* and *current hash* are used for write-serializability and the *chain hash* is used for freshness. The chained hash of an attestation is a hash over the data in the current attestation and the chain hash of the previous attestation.

$$\text{chain hash} = \text{hash}(\text{data}, \text{previous chain hash}) \quad (1)$$

It applies to both put and get attestations. The chain hash thus depends not only on the current attestation but also on the history of all the attestations for that block so far (because it includes the previous chain hash).

We say that that two attestations A and B *chain correctly* if the chain hash in B is equal to the hash of B's data and A's chain hash. The chain hash creates a cryptographic chain between attestations: it is computationally infeasible to insert an attestation between two attestations in a way that the three attestations will chain correctly. We can see that the client put attestation is the same with the integrity signature described in Section 4 and provided with every write.

The purpose of the nonces is to prevent the cloud from constructing cloud attestations ahead of time before a certain write happens. Then the cloud could provide those stale attestations to readers and thus cause them to read stale data. Because of the cryptographic chain induced in the attestations by the chain hash, the cloud cannot insert an attestation C between two other attestations A and B; thus, the cloud can give stale data to a client and pass the attestation audit test only if he produced and saved attestation C before he gave out B. A client gives a nonce to the cloud only upon making a request; the cloud cannot know this nonce before request C is made (the nonce is randomly chosen from a large field) so the cloud will not have been able to construct an attestation for this nonce ahead of time.

The structure of the attestation depends on the security level desired. If the user does not want write-serializability and freshness, the attestations are not needed at all. If the user does not want freshness, the chain hash can be removed.

The *attestation data* is the data in the put client attestation before the hash and signature are computed. Note that the integrity signature discussed in write access control (Sec. 4) and stored with each block in Figure 1 is the last client put attestation to that block.

5.2 Protocols for Exchange of Attestations

Get:

1. *Client:* Send the get request, block ID and a random nonce to the cloud.
2. *Cloud:* Prepare and send back the entire block (including metadata and the attached integrity signature) from Fig. 1, the new chain hash, and the cloud get attestation.
3. *Client:* Verify the integrity signature and the cloud attestation (it was computed over the data in the block, chain hash, and nonce). Do not consider the get finished until all these checks succeed.

Put:

1. *Client:* Send the entire new block (content and metadata) and the client put attestation.
2. *Cloud:* If the client's attestation is correct (new hash is a hash of block content, datagroup verification key verifies the signature), send back the chained hash and put attestation. Store the new block (together with the client attestation).
3. *Client:* Verify the attestation. Consider a write committed only when this check succeeds.

6 Confidentiality and Integrity

In this section, we describe the techniques we use to ensure confidentiality and integrity. These techniques have been extensively used in previous work so we will not linger on them; rather, we explain how to integrate them in our proof mechanism and formalize the guarantees they provide.

Confidentiality (C). As mentioned earlier, we achieve confidentiality by having the clients encrypt the content placed on the cloud. Note that even though the cloud cannot gain information from the encrypted data, the cloud can deduce some information from the access patterns of the users to the data (e.g., frequency of reading a certain block). There has been significant theoretical work on masking access patterns [15, 27], but efficient such pro-

tocols are yet to be found so we do not make an attempt to mask them.

Integrity (I). As mentioned in write access control (Section 4), each time a user puts a block on the cloud, he must provide a signed hash of the block. Similarly, each time a user reads a block, he checks the signed hash using the public verification key available in the key block. Note that the special blocks used to store keys are also signed in this way, so their integrity is assured in the same way that all blocks are.

Detection of I Violation: the integrity signature on a block does not match the block's contents.

Proof of Violation of I: the block with violated integrity and the attestation from the cloud.

Recall that the get attestation from the cloud contains a hash of the block content with the integrity signature and is authenticated with the cloud's signature. If this hash of the block content does not verify with the hash from the integrity signature, it means that the cloud allowed/performed an invalid write, and the attestation itself attests to the cloud's misbehavior.

A client cannot frame an honest cloud. When the cloud returns an attestation, it is signed by the cloud so the client cannot change the contents of the attestation and claim the cloud stored a tampered block. Also, if a client falsely claims that a different verification key should be used, the cloud can exhibit the owner's signed attestation for the key block. This suffices because the data block and the key block include the version of the verification key.

7 Write serializability and Freshness

To detect deviations from W and F, the data owner periodically *audits* the cloud. The owner performs the auditing procedure at the end of each epoch.

A successful audit for a certain block in an epoch guarantees that the cloud maintained freshness and write-serializability of that block during the particular epoch.

The data owner assigns to each block some probability of being audited, so an audit need not check every block in every epoch. If a block is very sensitive, the owner can assign it a probability of one, meaning that the block will be audited in every epoch. If a block is not very important, the owner can assign it a smaller probability.

We cannot hide the rate at which a block is audited, since the cloud can simply observe this. However, the cloud cannot be allowed to know exactly which epochs will feature a block's audit, since if it did, it could undetectably misbehave with regard to that block during other epochs. Users, in contrast, need to be able to figure out these epochs because they need to send cloud attestations to the data owner in exactly these epochs. Thus, we use a technique that ensures that only users who know the read access key for a block can determine the epochs in which it will be audited. Specifically, we audit a block whenever:

$$\mathsf{prf}_{\mathsf{read\ key}}(\mathsf{epoch\ number},\ \mathsf{blockID}) \mod N = 0,$$

where prf is a pseudorandom function [14], N is an integer included in plain text in the block metadata by the owner. If a probability of audit p is desired, the data owner can achieve this by setting $N = \lfloor 1/p \rfloor$. Note that while hashes are used in practice for the same purpose as we use a prf, hashes are not secure for such usage because their cryptographic specification only guarantees collision-resistance and not necessarily pseudorandomness, as needed here.

We do not try to prevent against users informing the cloud of when a block should be audited (and thus, the cloud misbehaves only when a block is not to be audited). Auditing is to make sure that the users get correct data and their puts are successful. If they want to change the data, they can do so using their access permissions and do not have to collude with the cloud for this. As mentioned before, the owner should ensure (via other means) that users do not use their access permissions on behalf of unauthorized people.

When a block is meant to be audited, clients send the owner the attestations they receive from the cloud. Clients do not need to store attestations. The owner separates these attestations by block and sorts them by version number. This generally requires little processing since the attestations will arrive in approximately this order. The attestations for the same version number are sorted such that each two consecutive attestations satisfy Equation (1); we envision that the cloud could attach a sequence number to the attestations to facilitate this sorting. If some clients do not send attestations because they fail or are malicious, they have no guarantees on whether they read correct data or their put got committed. All clients sending attestations will have such guarantees. The clients cannot frame the cloud by not sending attestations. The owner will ask the cloud to provide cloud attestations for any missing attestations in the sequence and an honest cloud will keep copies for the duration of an epoch. Alternatively, the cloud can only keep copies of the attestations' data without signatures, which it can reconstruct on-demand, thus storing less. If the cloud cannot provide these, the cloud is penalized for non-compliance with the auditing procedure.

Once the owner has the complete sequence of attestations, it performs checks for write-serializability and freshness, as we describe in the subsections below. After auditing, the owner and the cloud create a Merkle hash tree of the entire storage, exchange attestations that they agree on the same Merkle value, and *discard all attestations or attestation data* from the epoch that just ended. The Merkle hash can be computed efficiently using the hashes of the blocks that were modified and the hashes

of the tree roots of the blocks that were not modified in this epoch. The owner updates the family key block if there were any membership changes.

Interestingly, we will see that auditing only makes use of public keys. As such the owner can outsource the auditing tasks (e.g. to a cloud competitor).

Due to space constraints, the presentation of the theorems and proofs below is in high-level terms, leaving a rigorous mathematical exposition for a longer paper.

7.1 Write-serializability (W)

Requirement on the cloud. During the epoch, the cloud is responsible for maintaining the write-serializability of the data. That is, the cloud must make sure that every put advances the version number of the most recently stored block by exactly one. If a client provides a put for an old version number, the cloud must inform the client of such conflict. It is now the client's choice to decide what action to take: give up on his own change, discard the changes the cloud informs him of, merge the files, etc.

One attack on write-serializability is a *fork attack* (introduced in [20]). The cloud provides inconsistent views to different clients, for example, by copying the data and placing certain writes on the original data and other writes on the copy. If two clients are forked, they will commit two different updates with the same version number on the cloud.

A correct write chain of attestations is a chain of put cloud attestations where there is exactly one put for every version number between the smallest and the largest in the sequence. Moreover, the smallest version number must be one increment larger than the version number of the block at the beginning of the epoch.

Detection of W Violation: The sequence of put attestations do not form one correct write chain.

The following theorem clarifies why this statement holds. It assumes that users send all the write attestations they get from the cloud to the owner. If they do not send all these attestations, the theorem holds for every complete interval of version numbers for which attestations were sent.

Theorem 1. *The cloud respected the write-serializability requirement for a block in an epoch iff the cloud's put attestations form one correct write chain.*

Proof. First, let us argue that write-serializability implies one chain. If the cloud respects write-serializability, it will make sure that there are no multiple writes for the same version number and no version number is skipped; therefore, the attestations form a correct chain.

Now, let us prove that one chain implies write-serializability. A violation of this property occurs when a client performs an update to an old version of the data. Suppose the current version of the data on the cloud is n and the client is aware of $m < n$. When the client places a put, the version number he uses is $m+1 \leq n$. Suppose the cloud accepts this version and provides a cloud attestation for $m+1$. Since $m+1 \leq n$, another put with version $m+1$ committed. If that put changed the block in a different way (thus inducing a different new hash in the attestation), the owner will notice that the attestations split at $m+1$. □

Note that for W, the auditor does not check chain hashes and thus it does not need get attestations.

Proof of Violation of W: The broken sequence of write attestations as well as the cloud attestation for the current family key block.

This constitutes a proof because cloud attestations are unforgeable. A client cannot frame an honest cloud. A proof of violation consists of attestations signed by the cloud; thus the client cannot change the contents of the attestations and create a broken sequence.

7.2 Freshness (F)

Requirement on the cloud. During the epoch, the cloud must respond to each get request with the latest committed put content and compute chain hashes correctly (based on the latest chain hash given) for every cloud attestation.

A correct chain of attestations is a correct write chain with two additional conditions. First, the hash in each read attestation equals the new hash in the write attestation with the same version number. Second, the chain hash for an attestation and the chain hash of the previous attestation in the sequence satisfy Eq. (1).

Detection of F Violation: The attestations do not form one correct chain.

Theorem 2. *The cloud respected the freshness requirement iff the attestations form one correct chain.*

Proof. It is easy to see that if the cloud respected the freshness requirement, the attestations will form one chain. Each attestation will be computed based on the latest request.

Let us show that if the freshness requirement is violated, we do not have a correct chain. We proceed by contradiction assuming that we have a correct chain. There are two key points we will use. The first is that each chain hash the cloud gives to a client is dependent on some randomness the client provides. This is the random nonce for get or the new hash for put. The cloud cannot compute the chain hash before it has this randomness. The second point is that the value of a chain hash recursively depends on all the history of chain hashes before it.

Let A be an attestation, and B the attestation preceding it in the chain. Assume the cloud violated the freshness requirement when answering the request corresponding

o A, but the attestations form a correct chain for contradiction purposes. Thus, B was not the latest request performed before A; instead, let C be this request. Thus, he cloud gave out attestations B, C, and A in this order. C must come somewhere in the correct attestation chain. It cannot come after A because the chained hash of C will have to depend on the chain hash of A. Due o the client-supplied randomness and hardness assumptions on the hash function (random oracle), the cloud can compute the chain hash for A only when he gets the A equest which happens after the C chain hash was given out. If C comes before A in the sequence, it must come before B because we assumed that the freshness requirement was violated at the time of this request. This means hat B's chain hash depends on C's chained hash, which s not possible because the cloud would not know C's client-provided randomness when he has to answer to B. Therefore, we see that the chain cannot be correct when reshness is violated. □

Proof of Violation of F: The broken sequence of attestations as well as the cloud attestation for the current family key block.

A client cannot frame an honest cloud. A proof of violation consists of attestations signed by the cloud; thus, he client cannot change the contents of the attestations nd create a broken sequence.

7.3 Discussion

For efficiency, many storage servers allow gets to proceed in parallel, while serializing puts. CloudProof also allows the bulk of the get operations to happen concurrently, and serializes only the computation of the chain hash for gets to the same block. For example, consider multiple concurrent gets for the same block. These gets can proceed in parallel (e.g., contacting nodes in the data center, performing disk accesses, preparing result); the cloud only needs to make sure that updates to the chain hash for the same block do not happen concurrently. Updating a hash takes very little time, so the loss in concurrency is small.

CloudProof protects W, F, and I with respect to each block in part. In some cases, one might want these properties to be satisfied with respect to a collection of blocks, for example, so that different blocks are updated consistently. CloudProof can satisfy this requirement by viewing the block collection of interest as one block and having one attestation for this collection of blocks. CloudProof will serialize accesses to this collection of blocks in the same way as for one block.

The protocol for freshness requires the cloud to update a chain hash and store some metadata upon any get operation, so a malicious client unauthorized to read the data can DoS a server computationally and storage-wise. A solution to this attack is to require each client to provide a correct "client get attestation" signed with the read access key to the cloud when performing a get. An evaluation of this scenario is future work.

8 Implementation

We implemented CloudProof on top of Microsoft's Windows Azure [22] cloud platform. Our implementation consists of about 4000 lines of C#. CloudProof only relies on a get/put and sign/verify interface from the cloud, which makes it easy to adapt to other cloud systems.

Background on Azure. First, we give the needed background on Azure [32]. Azure contains both a storage component and a computing component. CloudProof uses the blobs and queues storage services. Blobs are key/value stores mapping a blob identifier to a value. Queues are used for reliable messaging within and between services.

The compute component consists of *web* and *worker* roles, which run on virtual machines with different images of Windows Server 2008. The web roles are instances that can communicate with the outside world and receive HTTP requests. The worker roles are internal running instances which can communicate with the web roles (via storage nodes) and access storage. There is no guarantee whether the worker, web or storage nodes are collocated on the same machine. Furthermore, the storage and compute components are provided as different services so they may be located in different data centers, although Azure allows us to specify an affinity group for our storage and compute nodes.

CloudProof consists of four modules. The *data user/client* is a client-side library that exports get and put interface. A data user uses this library to perform get and put calls to the cloud. It exchanges blocks and attestations with the cloud. The *cloud* runs on top of Azure and responds to get and put requests and exchanges attestations with the client. The *data owner/enterprise* runs on a data owner's premise. This is a library to be used by the data owner. It serves to add or remove permissions to users or groups and for auditing. It interacts with the cloud component to update family key blocks. If the data owner wants to get or put data blocks, it needs to create a client instance for itself. As mentioned, the owner can outsource the auditing task to another party, the *auditor*. It collects attestations and audits them according to the algorithms in Section 7.

Let us explain how a request processing proceeds. The clients and the owner send HTTP requests to the web roles, which then place the requests in a queue, together with a blobID. The workers poll the queue for requests to process. The worker roles place the response into the blob with the given blob ID. Web roles poll response blobs periodically to get the reply for the request they

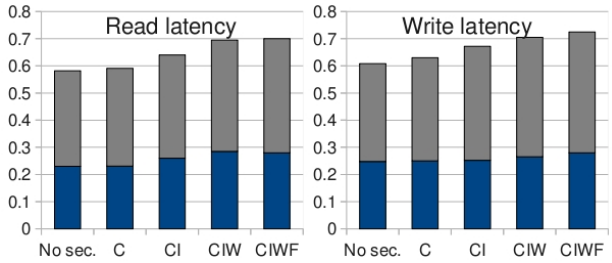


Figure 3: End-to-end and effective read and write latency. Each bar indicates the end-to-end latency incurred, with the lower section showing the effective latency.

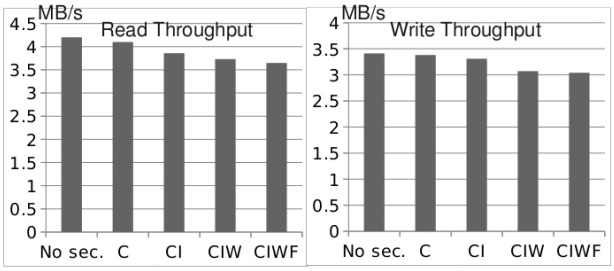


Figure 4: Effective read and write throughput.

made.

The cryptographic algorithms used are the .NET implementations of SHA-1 for hashing, AES for symmetric encryption and 1024 bit RSA for signing. Any of these schemes can be easily substituted with more secure or faster ones.

9 Evaluation

In this section, we evaluate the performance of CloudProof. We investigate *the overhead CloudProof brings to the underlying cloud storage system without security* to establish whether the security benefits come at a reasonable cost. We are not interested in examining or optimizing the performance of Azure itself because CloudProof should be applicable to most cloud storage systems.

CloudProof targets large enterprise storage systems, source code repositories, and even small, personal file systems. These tend to be applications that can tolerate larger client-cloud latency (which is an inherent result of the different geographic locations of various clients/organizations with respect to the cloud).

The design of CloudProof allows *separability to five security modes*, each corresponding to the addition of a new security property: no security, just confidentiality (C), confidentiality and integrity (CI), the previous two with write-serializability (CIW), and full security (CIWF). We will see how performance is impacted by adding each security property. For these experiments, the client-side machine is an Intel Duo CPU, 3.0 GHz, and 4.0 GB RAM.

Microbenchmarks. We observed that a large factor of performance is given by the latency and bandwidth between the client and the cloud's data center (which seems to be on the other coast from the data we get), which do not depend on our system. Therefore, we will also include "effective" measurement results which are computed by subtracting the client-cloud round-trip time from the end-to-end latency.

To evaluate latency, we perform 50 reads and 50 writes to different blocks (4 KB) and compute the average time per operation. Figure 3 shows our latency results. We can see that the overhead increases with every addition of a security requirement. The overhead added by W is caused by the creation, verification and handling the attestations; adding a chain hash for F causes a small overhead increase. The overhead of our scheme (CIWF) as compared to the no security case is $\approx 0.07s$ which corresponds to 17%. With respect to confidentiality only (minimal security), CIWF is $\approx 14\%$.

Let us understand which components take time. Each request consists of client preprocessing (0.5%), network round trip time between server and client (36%), cloud processing (62%), and client postprocessing (1.5%). We can see that most time is spent at the server. This is because the web and worker roles communicate with each other using the storage nodes, and all these nodes are likely on different computers. We specified affinity groups to colocate nodes close to each other, but we do not have control over where they are actually placed.

Figure 4 shows the effective throughput incurred in our system. The graphs are generated by reading and writing a large file (sent to one cloud worker), 50 MB, and dividing its size by the effective duration of the request. We generated effective measurements because the throughput between the client and the cloud was a bottleneck and the results were similar for and without security (almost no overall overhead). We can see that the throughput overhead (due to cryptographic operations mostly) for write is 11% and for read is 12%, which we find reasonable.

An important measure of scalability is how the system scales when workers are added. We obtained a VIP token for Azure that allows creation of 25 workers (given the early stage of Azure deployment, it is hard to get more workers for free). We spawn 25 clients that can run in parallel and each such client sends two requests to the cloud (a request is sent only after the previous request finished); this experiment is repeated many times. We measure how many pairs of requests are satisfied per second as we increase the number of worker roles. Figure 5 shows our results. We can see that each pair of requests is satisfied in about 1s which makes sense given the latency results presented above. We can see that the scaling is indeed approximately linear. This makes sense because we designed our security protocols to allow all requests to proceed in parallel if they touch different blocks.

Moreover, for scalability, our access control scheme

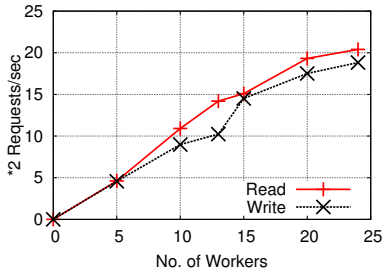


Figure 5: Pairs of requests executed per second at the cloud as depending on the number of workers.

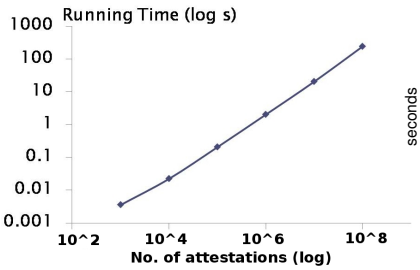


Figure 6: Auditing performance for one owner node.

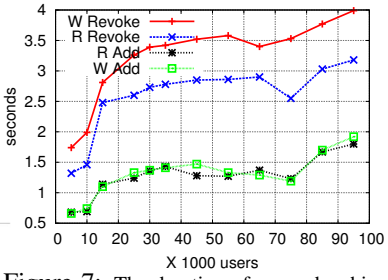


Figure 7: The duration of a membership update as depending on the number of members with access to a block family. R denotes read and W denotes write.

should be able to support efficiently many members and blocks. Our scheme has the benefit that revocation and addition do not depend on the number of blocks in a family. It only depends on the square root of the number of members with access to the block family. As we can see in Figure 7, the square root factor is almost irrelevant; the reason is that the constants multiplying the square root are much smaller than the constant latency of read and write. We can see that for large enterprises with $100,000$ employees, group addition and revocation remain affordable even if every single employee is present on an access control list.

A block family is created when a block's ACL is changed to an ACL that no other block has. The owner must put a new family key block on the cloud. If the membership change from the previous ACL (if one existed) to the current one is not large, the owner can just copy the family key block for the previous ACL and perform the appropriate revocations and additions. Otherwise, the owner should choose new keys, compute broadcast encryptions to the new members, and put the resulting data on the cloud. For an ACL of 1000 new members, these operations sum up to $\approx$ 20s. Any blocks added to the block family will not require key block family changes. The block just needs to be encrypted with the proper key.

Figure 6 shows the performance of auditing. For example, it takes about 4 minutes to verify 10^8 attestations. This number corresponds to a large company of 10^5 employees each making about 1000 changes in an epoch of a month. Furthermore, auditing can be easily parallelized because it just involves checking each consecutive pairs of attestations in a chain so the chain can be split in continuous sections, each being distributed to a different core. In fact, auditing can be performed on an alternate cloud, which, due to market competition, will have incentive to run the audit correctly. The duties of the owner are just auditing and membership changes, so we can see that, even for enterprise storage, a lightly used commodity computer would be enough to handle all of the owner's duties.

Storage Overhead. The total overhead per block in CloudProof is 1120 bits if we use RSA signatures or 256 bits if we use short signatures (Boneh-Lynn-Shacham). A block in Azure may be arbitrarily large, so this overhead may be a small percentage of large blocks. The family key block consists of $1120 + 1024 * \sqrt{n} + 1024 * n$ bits, where n is the number of users in a family. Had we used a more efficient broadcast encryption scheme, the linear factor would be removed. In a commercial source repository trace (presented below), we found that the maximum value of $\sqrt{n}$ was 14 and that there were 30 total groups. Therefore, the storage overhead for family key blocks is less than 26.4 KB for the lifetime of the trace. All attestations are about 1300 bits (or about 400 bits with a short signature). The cloud only needs to keep the latest put client attestation for each block and the unsigned attestation data ($\approx$ 258 bits) for all attestations in an epoch for auditing purposes.

Macrobenchmarks. To determine how many users are in real enterprise groups and to understand the frequency of access control changes, we obtained traces of group membership changes in a version control repository for a very large widely used commercial software (whose name we cannot disclose) that has more than 5000 developers. From these traces, we computed the dates of all *revocation events*: changes to group membership where at least one member was deleted from the group. For each event, we computed the size of the group after deletions of members. As described, the square root of this group size controls the time required to compute new keys for the block family. As we showed in Figure 7, our system can compute new keys for groups of $100,000$ users in less than 4 seconds. In particular, we found that computing keys for all revocation events in a month, assuming all groups in our trace have 250 members, took an average time of less than 1.6 seconds. This shows that our system is capable of easily handling the group sizes and frequency of revocations in this real application.

We looked at the commit histories for two large open source projects hosted on Github: Ruby on Rails and Scriptaculous. Both projects are widely used and under active development. Source code repositories require integrity guarantees: adversaries have broken into such

	week max	week avg	month max	month avg
RoR	55.7	9	93	38
ST	4.2	0.91	8.2	2.4

Table 1: Maximum and average storage requirements in KB per epoch for all epochs with at least one client put request. RoR stands for "Ruby on Rails," while ST stands for "Scriptaculous."

repositories in the past to corrupt software which is then widely distributed. Distributed development also benefits from F and W.

To create benchmarks, we looked at the history of all commits to all files in Ruby on Rails for six months from July 2009 to November 2009, and all files in Scriptaculous for one year from November 2008 to November 2009. Reads are unfortunately not logged in the traces; however, from microbenchmarks, we can see they have a similar overhead to writes. We used the history provided in each repository to identify the size of each file after each commit and the identity of the committer. We then replayed this history, treating each commit as a separate put whose key is equal to the file name and with a value of the appropriate size. Figure 8 shows the results for the both traces. The overall overhead for Ruby on Rails is 14% and for Scriptaculous is 13% for providing all security properties. These results show that we can achieve our properties with modest overhead, and that our microbenchmarks are a good predictor of our overhead for these applications.

For storage overhead, we first computed the number of distinct files in each repository in the state it was at the end of the trace. For Ruby on Rails we find 5898 distinct files, totaling 74.7 megabytes. At 1120 bits of overhead per file, CloudProof requires 806 KBs of storage for metadata, an overhead of roughly 1.1%. For Scriptaculous we find 153 distinct files, totaling 1.57 MB, yielding a storage overhead for CloudProof of 1.3%.

We then evaluated the cloud storage required to hold attestations from clients at different epoch lengths for both traces. We considered the extreme case where the epoch is equal to the duration of the trace: i.e., the cloud keeps all attestations from all clients. For the Ruby on Rails trace we have 71241 total client puts, which requires 2.2MB of storage on the cloud for 258 bits per attestation. For the Scriptaculous trace, we have 2345 total client puts, which requires 73KB of storage. We then looked at two cases: the epoch is one week and 30 days. Table 1 shows the results. We see that the amount of storage required for attestations in both traces is low, under 100KB, even for a long epoch time of 30 days.

The choice of epoch is a tradeoff between cloud storage overhead and audit time. Our trace results show that for the source repositories we considered, even an epoch length of six months to a year requires modest overhead. Shorter epochs require less than 100 kilobytes of storage. This appears reasonable, especially given that the storage for attestations may be append-only and need not be highly available. Other applications, of course, may require more puts during an epoch and therefore incur more overhead for storage. With our current implementation, however, one gigabyte of storage is sufficient to hold attestations for over 33 million puts; we believe this should be sufficient to allow a reasonable epoch for most applications. As we showed, the time to audit is not a gating factor for the length of an epoch, because audits can be carried out in a short amount of time.

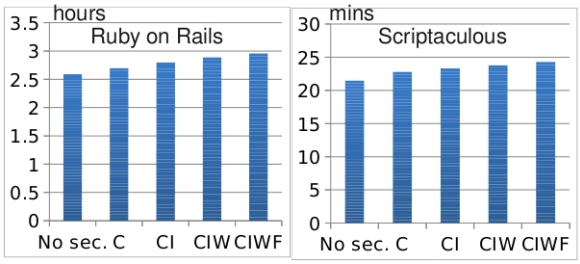


Figure 8: Runtime for macrobenchmarks.

From the results above, we conclude that CloudProof has a reasonable overhead and is practical for real storage applications with security needs.

10 Related Work

The related work can be divided in four parts.

Existing Cloud Systems. Today, there exist a plethora of cloud storage systems (Amazon S3, Google BigTable, Microsoft Azure, Nirvanix CloudNAS, etc.); however, these systems do not guarantee security. The cloud system that provides most security, to the best of our knowledge, is Tahoe [31]. It is a peer-to-peer file system that allows users to check data integrity and provides access control. However, Tahoe does not detect violations to W and F, nor does it provide proofs of violation.

Secure File and Data Storage Systems include related systems [4], [3], [10], [18], [20]. In comparison to these systems, CloudProof brings the three main contributions listed below. These systems do not provide these properties, not because of shortcomings in their design, but because they were designed for a different setting than the cloud: mostly for personal storage hosted on some remote untrusted and not liable servers. We also argue that they are not easily extendable to achieve our desired properties, which illustrates that there was a need for a new design for the cloud setting.

(1) They detect, but *do not prove* server misbehavior and *do not keep the client accountable to the server*. CloudProof provides such guarantees to enable a financial contract between the client and the cloud regarding security.

(2) CloudProof maintains the *high scalability* of the cloud because it is a crucial aspect of the cloud promise and large enterprises are important customers of the cloud. Our access control scheme is especially scalable.

(3) CloudProof provides detection (and proofs) to both *write-serializability and freshness*. This is because an enterprise affords to dedicate a compute node for auditing and membership changes. Most previous work cannot detect violations to both properties.

SiRiUS [10] is perhaps the most related previous work to CloudProof. In SiRiUS, each user stores his file system on a remote untrusted server and can detect integrity violations to his data. SiRiUS does not guarantee write serializability: two users can read a file at the same time, place updates subsequently with the second user ignorantly overwriting the first user's update. SiRiUS does not offer freshness as we define it in Section 2. It offers a weaker type of F: fetched data can be stale if it is not older than a certain time interval. Furthermore, SiRiUS does not scale to enterprise sizes. Each user has a separate file system that also contains files from other user's file systems for which the user has write access. When checking the freshness of a file upon read, users need to check the file system of each user with write access, verify a Merkle tree for that access, and decide who has the newest version. In an enterprise setting, some files can be accessed by thousands of users so this approach would not scale. Moreover, to ensure freshness, SiRiUS requires each user to re-sign the top of his Merkle hash every few minutes or seconds. This is not a reasonable assumption we believe, because the users are not necessarily online all the time. Moreover, in SiRiUS, users can defeat their access permissions. Unauthorized users can delete data or seize unattained permissions by overwriting the metadata of some files with their own. These attacks occur because Sirius's premise is not to change the server software and allow it to run on any remote file system implementation such as NFS. In contrast, cloud providers can always install software of their choice on their nodes if they want to provide security. Finally, SiRiUS does not provide proofs needed in our cloud setting.

Plutus [18] uses key rolling, but every time a user realizes that the key has changed, he contacts the owner to ask the key. This approach would demand more resources from an enterprise to satisfy the requests of all their employees. We combine key rolling with broadcast encryption and the notion of block families to achieve key distribution without involving the owner/enterprise.

SUNDR [20] is a secure file system that offers fork consistency by having clients check histories of snapshots of file versions (VSLs). First, SUNDR does not provide read access control and it is not clear how to enhance it with scalable read access control. Our key distribution protocol uses broadcast encryption and key rolling to support fast and scalable read access, and handles user read access revocation efficiently. Second, a straightforward addition of W to SUNDR, would be unscalable. If SUNDR sends all VSLs to the owner for auditing, SUNDR could achieve write-serializability easily because the owner would notice a fork attack. However, each VSL entry includes version numbers for all files in the file system, and there can be many such files. In contrast, CloudProof's attestations only contain information about the block accessed. On the other hand, one positive aspect of SUNDR in this regard is that it can order updates across files, whereas CloudProof can only order updates within a file. We made this tradeoff for scalability. Third, CloudProof provides improvements regarding freshness over SUNDR. Even with auditing at the owner, SUNDR would not achieve freshness for clients that are only readers. The server can give a client a prefix of the current history, thus not informing him of the latest write. Since the reader will not perform a write, a fork will not occur. SUNDR can be easily extended to provide proofs of integrity violation, but providing freshness violation proofs seems harder. Finally, SUNDR does not scale to enterprise-sizes because of the long history chain of signatures that clients must check for every fetch. For highly-accessed files or many users and files, version snapshots can grow large and many.

Cryptographic Approaches. Kamara and Lauter [19] investigate cryptographic techniques useful in the cloud setting. They mention proofs of data possession or retrievability (POR) [17], [2], [25], [9], which allow a server to prove to the owner of a file (using sublinear or even constant space usage) that the server stores the file intact (the file has integrity) and can retrieve it. HAIL [6] allows a distributed set of servers to prove file integrity and retrievability. [29] allows updates and enables public verifiability of the integrity of the data at the cloud.

Such work is very useful for proving integrity of archived data; however, it is not sufficient as a holistic cloud systems solution. The fact that the file is correct and retrievable on the cloud does not mean that the cloud will respond with correct data upon a request. For example, suppose that the user requests the cloud to prove that the file is intact and the cloud successfully does so. Then, a few users request various blocks of the file; the cloud can return incorrect data (junk or stale) and there is no mechanism in place for checking this. Having each client ask the cloud for a POR before a get is too expensive. Also, most of these schemes deal with archived data and are not efficient on updates. They either require some non-negligible overhead of preprocessing when placing a file on the server [17] or that all updates be serialized [29] and thus have poor scalability. Moreover, they do not provide write-serializability or freshness (and thus, no proofs of violations for these properties). For instance, the cloud can overwrite updates and retrieve stale data because the integrity checks will not fail. Lastly, these schemes do not provide access control and are not designed for many concurrent accesses by different users.

Byzantine Fault Tolerance (e.g., [7]) proves correctness of query execution at remote server replicas given that the number of Byzantine (faulty, malicious) replicas is at most a certain fraction. However, this approach is not applicable to the cloud setting because all the nodes in a data center belong to the same provider. If the provider is malicious, all the nodes are malicious. Furthermore, most nodes are likely to be collocated geographically and run the same distribution of software and likely crash from similar factors. One idea is to use BFT with multiple cloud providers. This approach will indeed decrease the chance of security problems; however, clients will have to pay all the cloud providers, and, if the data gets lost at all parties, the client has no remuneration assurance. Chun et al. [8] also uses the concept of chained attestations, which they store in a trusted-hardware log; their goal is to prevent equivocation by Byzantine clients and ultimately improve performance of commit.

Secure Audit Trails and Logs. Research in secure digital audits aims to verify the contents of a file system at a specific time in the past. For example, in [24], a file system commits to the current version of its contents by providing a MAC on its contents to a third-party. At a later time, an auditor can check that the file system still contains the old version using the MAC token.

There has also been work on secure logging [30], [26]. In this work, a trusted machine writes encrypted logs that cannot be read or modified undetectably by an outsider. This work does not consider a large number of users concurrently accessing the data, there is no read and write access control (one key allows both read and write), the log is typically just appendable and it is not optimized for writing in the middle, and a malicious outsider manipulating the order in which updates and reads are performed on the logging machine can compromise W and F.

Moreover, in all this work, the owner cannot convince a third party of some security violation.

11 Conclusions

We propose proofs of security violations for integrity, write-serializability and freshness as a tool for guaranteeing security in SLAs. We build a secure cloud storage system that detects and proves violations to these properties by combining cryptographic tools in a novel way to obtain an efficient and scalable system. We demonstrate that CloudProof adds reasonable overhead to the base cloud service.

References

[1] AMAZON. Amazon s3 service level agreement, 2009. http://aws.amazon.com/s3-sla/.

[2] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. Provable data possession at untrusted stores. In *ACM CCS* (2007).

[3] BINDEL, D., CHEW, M., AND WELLS, C. Extended cryptographic filesystem. In *Unpublished manuscript* (1999).

[4] BLAZE, M. A cryptographic file system for unix. In *ACM CCS* (1993).

[5] BONEH, D., GENTRY, C., AND WATERS, B. Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys. *Lecture Notes in Computer Science* (2005).

[6] BOWERS, K. D., JUELS, A., AND OPREA, A. HAIL: A High-Availability and Integrity Layer for Cloud Storage. *CCS* (2009).

[7] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS* (2002).

[8] CHUN, B.-G., MANIATIS, P., SHENKER, S., AND KUBIATOWICZ, J. Attested Append-Only Memory: Making Adversaries Stick to their Word. *SOSP* (2009).

[9] DODIS, Y., VADHAN, S., AND WICHS, D. Proofs of Retrievability via Hardness Amplification. *TCC* (2009).

[10] E.-J. GOH, H. SHACHAM, N. M., AND BONEH, D. Sirius: Securing remote unstrusted storage. In *NDSS* (2003).

[11] FERDOWSI, A. S3 data corruption?, 2008. http://developer.amazonwebservices.com/connect/thread.jspa?threadID=22709&start=0&tstart=0.

[12] FIAT, A., AND NAOR, M. Broadcast encryption. *Proc. of Crypto* (1993).

[13] FU, K. Cepheus: Group sharing and random access in cryptographic file systems. Master's thesis, MIT, 1999.

[14] GOLDREICH, O., GOLDWASSER, S., AND MICALI, S. How to Construct Random Functions. *JACM* (1986).

[15] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *J. ACM* (1996).

[16] Cloud security still the biggest concern/hurdle for google, microsoft, verizon. www.taranfx.com/blog/.

[17] JUELS, A., AND KALISKI, B. PORs: Proofs of retrievability for large files. In *ACM CCS* (2007).

[18] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *USENIX FAST* (2003).

[19] KAMARA, S., AND LAUTER, K. Cryptographic cloud storage. In *ACM Workshop on Cloud Security* (2009).

[20] LI, J., KROHN, M., MAZIERES, D., AND SHASHA, D. SUNDR: Secure untrusted data repository. In *OSDI* (2004).

[21] MICALI, S., RABIN, M. O., AND VADHAN, S. P. Verifiable Random Functions. In *IEEE FOCS* (1999).

[22] MICROSOFT CORPORATION. Windows Azure. http://www.microsoft.com/windowsazure.

[23] MICROSOFT CORPORATION. Windows Azure Pricing and Service Agreement, 2009. http://www.microsoft.com/windowsazure/pricing/.

[24] PETERSON, Z. N. J., BURNS, R., AND ATENIESE, G. Design and Implementation of Verifiable Audit Trails for a Versioning File System. *USENIX FAST* (2007).

[25] SACHAM, H., AND WATERS, B. Compact Proofs of Retrievability. *ASIACRYPT* (2008).

[26] SCHNEIER, B., AND KERSEY, J. Cryptographic Support for Secure Logs on Untrusted Machines. *USENIX Security* (1998).

[27] SION, R., AND CARBUNAR, B. On the computational practicality of private information retrieval. In *NDSS* (2007).

[28] THOMSON, I. Google docs leaks private user data online, 2009. http://www.v3.co.uk/vnunet/news/2238122/google-docs-leaks-private.

[29] WANG, Q., WANG, C., LI, J., REN, K., AND LOU, W. Enabling Public Verifiability and Data Dynamics for Storage Security in Cloud Computing. *ESORICS* (2009).

[30] WEATHERSPOON, H., EATON, P., CHUN, B.-G., AND KUBIATOWICZ, J. Antiquity: Exploiting a Secure Log for Wide-area Distributed Storage. *ACM SIGOPS* (2007).

[31] WILCOX-O'HEARN, Z., AND WARNER, B. Tahoe - the least authority file system, 2008. http://allmydata.org/~zooko/lafs.pdf.

[32] WINDOWS AZURE PLATFORM. Azure SDK, 2009. http://msdn.microsoft.com/en-us/library/dd179367.aspx.

jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components

Carsten Weinhold, Hermann Härtig
Technische Universität Dresden, Germany
{weinhold,haertig}@os.inf.tu-dresden.de

Abstract

The Virtual Private File System (VPFS) [1] was built to protect confidentiality and integrity of application data against strong attacks. To minimize the trusted computing base (i.e., the attack surface) it was built as a stacked file system, where a small isolated component in a microkernel-based system reuses a potentially large and complex untrusted file system; for example, as provided by a more vulnerable guest OS in a separate virtual machine. However, its design ignores robustness issues that come with sudden power loss or crashes of the untrusted file system.

This paper addresses these issues. To minimize damage caused by an unclean shutdown, jVPFS carefully splits a journaling mechanism between a trusted core and the untrusted file system. The journaling approach minimizes the number of writes needed to maintain consistent information in a Merkle hash tree, which is stored in the untrusted file system to detect attacks on integrity. The commonly very complex and error-prone recovery functionality of legacy file systems (in the order of thousands of lines of code) can be reused with little increase of complexity in the trusted core: less than 350 lines of code deal with the security-critical aspects of crash recovery. jVPFS shows acceptable performance better than its predecessor VPFS, while providing much better protection against data loss.

1 Introduction

Both VPFS and its successor jVPFS are built in response to the observation that the enormous code bases of monolithic OSes (hundreds of thousands to millions of lines of code) are likely to contain exploitable weaknesses that jeopardize platform security. Apparently, this observation is valid especially for mobile devices that currently have the highest speed of hardware technology innovation. Almost daily reports, for example on successful attacks on core system components such as drivers [2], USB stacks [3], passcode protection [4], common applications such as text messaging [5] or on "jailbreaks" [6], which consistitute successful attacks, too, substantiate that claim of significant vulnerability. On the other hand, as smartphones, tablets and similar appliances have evolved into powerful and versatile mobile computers, professional users are starting to use them for critical data. For example, a doctor making house calls may use such a device to store patient records, which are not only sensitive from the patient's point of view, but also subject to legal requirements. Or a mobile device may store documents that are classified or contain trade secrets. Mobile payment systems on the other hand have strong integrity requirements to prevent tampering. Yet mobile devices are frequently connected to insecure networks (public WiFi, etc.) and in certain situations, users even must hand them over to untrusted third parties (e.g., leave them at the reception when visiting a company).

A general approach that so far seems mostly attractive for safety critical systems and to the miltitary is based on small isolation kernels or microkernels. Such kernels strongly separate applications, but also operating system components. Some of them [7], then called "hypervisors", contain the basic functionality to support virtual machine (VM) monitors and legacy OSes as guests. Based on such kernels, critical applications can run in their own compartments (built on microkernel services or in their own VMs) that are protected even against successful attacks on drivers or other parts of large, insufficently secure legacy OSes. Related work [8, 9] has also shown that applications can be split such that their security-critical cores run isolated, but reuse untrusted parts of the system for their non-critical functionality, thereby reducing the trusted computing base (TCB) of these applications by several orders of magnitude.

VPFS and jVPFS: File Systems for Microkernels. In previous work on VPFS [1] we built a file-system stack

that leverages a microkernel-based isolation architecture to achieve better confidentiality and integrity protection of application data. We achieved this by splitting the file-system stack into a small trusted and a larger untrusted part that reused Linux file-system infrastructure. Only the former is within the file-system TCB (see Figure 1 for an architectural overview). As VPFS uses untrusted components, which might be penetrated or otherwise corrupt data, integrity guarantees can only cover tamper evidence: manipulated data is detected through checksum mismatches and never delivered to applications. Unfortunately, better integrity protection also reduces availability of file-system data in the event of an unclean shutdown; for example, checksums may no longer match the corresponding file contents, if the battery of the mobile device failed unexpectedly or the system crashed.

With jVPFS, we address the problem of ensuring both robustness and integrity in a split file-system stack as described above. In monolithic file system stacks, the code for ensuring consistency of on-disk structures (e.g., journaling, soft updates [10]) is rather complex and difficult to get right [11, 12]. Recent research [13, 14, 15] has shown that even file system implementations that are widely used in production environments still have bugs, commonly found in code paths used for error handling and post-crash recovery. A subset of these bugs are security critical [2]. It is therefore a primary design goal for us to keep the inherent complexity of consistency mechanisms out of the file-system TCB in order to lower the risk of introducing exploitable design and implementation errors. Nevertheless, existing file system implementations are well tested and sufficiently reliable in common application scenarios (when not subject to sophisticated attacks). For practical reasons, it is therefore desirable to reuse this infrastructure in order to reduce engineering effort.

Contribution. The work presented in this paper makes the following contributions: We extend the file-system TCB for *confidentiality*, *integrity*, and *freshness* of all data and metadata such that these protection goals can be reached even after an unclean shutdown. To this end, we identify and isolate the security-critical functionality required to recover a consistent file system after a crash and discuss how existing, untrusted consistency infrastructure can be reused to complement the security-critical part. We devise a novel cooperation scheme that lets trusted and untrusted components cooperate and discuss precisely which metadata information must be revealed to untrusted code in order to facilitate this cooperation. We evaluate a prototype implementation.

Synopsis. On the following pages, we first provide required background on our security model and then present our design and optimizations in Sections 3 and 4, respectively. We evaluate our prototype in Section 5, before discussing related work in Section 6.

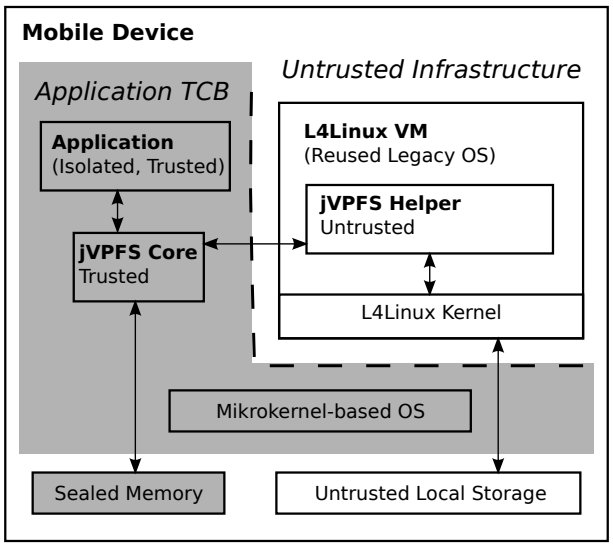


Figure 1: *jVPFS in a decomposed system architecture with strong isolation among components.*

2 Background

Figure 1 gives an overview of the system architecture into which the split jVPFS stack integrates. Strictly following the principle of least privilege, file-system contents are accessible only to the specific application that owns them. The small file-system kernel of jVPFS implements all security-critical functionality and reuses the file system stack provided by an untrusted, virtualized legacy OS. That is, Linux performs all non-critical tasks to manage persistent storage.

2.1 Security Model

The jVPFS security model is identical to the one for the original version of VPFS [1]; we summarize it here. We consider a strong attacker who is trying to compromise *confidentiality*, *integrity*, or *freshness* of critical data stored in a VPFS file system. Confidentiality means that, after authorization, the user can access his data only through a specific application. Our notions of integrity and freshness apply to both user data (file contents) and metadata (filenames, sizes, timestamps, etc). The file system provides only complete and correct data and metadata to the application. We require that VPFS can detect any tampering and whether data and metadata are up-to-date, preventing an attacker from rolling back file-system contents to an older version without being no-

ticed. We assume that both software and certain hardware attacks are possible. At run time, VPFS relies on hardware address spaces and virtual-machine boundaries in order to isolate the trusted and untrusted components effectively. To counter offline attacks, VPFS requires the mobile device to enforce a secure startup process of the application TCB and a small amount of access-restricted, tamper-resistant memory to store cryprographic keys and a checksum for integrity checks.

Software-based Attacks. Both software and data stored in the mobile device may be tampered with. Given the high complexity and enormous code size of the virtualized legacy OS, we must assume that the attacker can fully compromise it; hence, untrusted components may stop working correctly at any time. Nevertheless, we assume that in the common case, when not being attacked, they function as expected and cooperate with the trusted part of VPFS. In the case of jVPFS, the untrusted infrastructure is expected to store cryptographically protected file-system contents persistently, taking any necessary consistency constraints into account.

Components within the TCB are considered to be significantly harder to attack, because their isolated codebases present a smaller attack surface. We assume that they either work correctly, or not at all, should the secure boot process of the mobile device detect that their executable files or configuration have been tampered with.

Hardware-based Attacks. We assume that an attacker is able to directly access or manipulate the device's mass storage (e.g., a flash memory card), but he cannot successfully read or manipulate the contents of the tamper-resistant memory or break the secure boot process. To ensure tamper resistance, the device could be equipped with a trusted platform module (TPM) [16] or a small amount of secure flash memory that is directly integrated into the system-on-chip (SoC) package. Access to the secure flash memory must be restricted to certain software stacks by means of secure boot, possibly augmented with hardware-based access control as enabled by the ARM TrustZone [17] technology.

Secret keys stored in a TPM can be extracted with equipment that costs in the order of hundreds of thousands of dollars, but the process is destructive. Similarly, we assume that gaining direct access to the secure flash in the SoC is too hard for the attacker. Making a user-provided secret such as a PIN code part of the storage encryption key limits benefits of such an attack further.

2.2 Cryptographic Protection

Earlier work on cryptographic storage systems (e.g., [18, 19]) shows how file-system contents can be protected against offline attacks by using encryption. In jVPFS, AES-CBC encryption ensures confidentiality at the block level, thereby enabling efficient partial updates.

The state of the art technique for efficiently ensuring integrity and freshness is to use a Merkle hash tree [20]. By construction, the tree provides all necessary information to verify correctness and completeness of file-system contents; one can guarantee freshness by storing the root hash of the tree in tamper-resistent, persistent memory. In our system, the entire file system including its metadata (names, etc.) is protected by the Merkle tree. Simpler approaches without using a tree structure do not meet our requirements: Single hashes for large regions or entire files make partial updates expensive; furthermore, storing one independent hash per region or file requires impracticably much tamper-resistant storage. Using keyed hashes (e.g., HMACs [21]) instead makes hashed data vulnerable to roll-back attacks, thereby defeating freshness.

3 Design

In jVPFS, the Merkle hash tree is essential to strong integrity and freshness guarantees. It must always be possible to restore it to a consistent state. The design of the jVPFS consistency mechanism is driven by the principle of least privilege, aiming at a minimal attack surface of the implementation in order to increase its trustworthiness. There are two key challenges to reaching these goals:

1. **Making the Cut.** Part of our integrity requirement is that data provided to applications is complete; freshness demands that the latest file-system state is available. The consistency mechanism thus has security-critical functionality that must be identified and isolated from uncritical, potentially untrusted components in an efficient and effective way.

2. **Secure Cooperation.** Despite isolation, the two parts of the file-system stack must be able to cooperate. Therefore, at least some information describing consistent sets of updates must be revealed to and processed by untrusted components whithout jeopardizing confidentiality, integrity, and freshness of file-system state.

3.1 Consistency Paradigm

After a crash, the on-disk structures in the untrusted storage must contain enough information to restore the Merkle hash tree spanning the file system data and metadata.

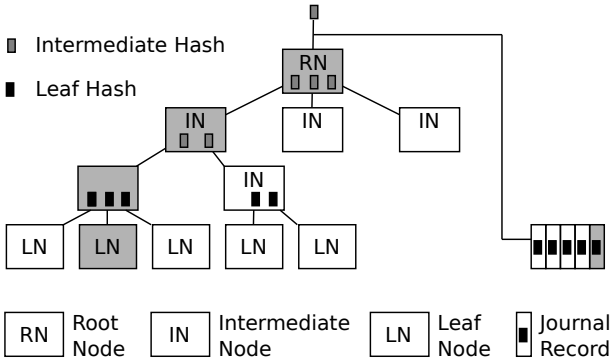


Figure 2: *Updating the Merkle hash tree: modification of one leaf node requires the complete chain of intermediate nodes up to and including the root node to be updated. Instead, it is more efficient to append leaf-node hash sums to a growing journal, which is cryptographically bound to the root hash.*

Naive Approach. The naive approach to meet this requirement is to ensure that the untrusted legacy file system always holds a consistent snapshot of the Merkle tree. In such a hash tree, changes are first made to the leaf nodes and then propagate to the root. Thus, modifications of the on-disk file system (e.g., writing new data to a file) must always include updates of all intermediate Merkle tree nodes up to the root node. This set of updates to the on-disk user data (i.e., file contents) and the Merkle tree nodes has to be applied *atomically*. In this context, *atomic* means that, after a system crash, either the complete set of updates is persistent, or the previous state is accessible—otherwise the hash sums become inconsistent, breaking the authentication chain. Figure 2 illustrates these update dependencies.

Unfortunately, atomic updates to Merkle trees are expensive, because small modifications such as writing a single block of user data involve writing as many tree nodes (i.e., disk blocks) as there are levels in the tree. For each of these blocks the trusted part of jVPFS has to perform cryptographic operations, further increasing run-time overhead and energy consumption. Also, modifying file contents might require updates of metadata such as file-size information, which is stored in an inode that must be protected by the Merkle tree, too. Thus, the costs of consistent, atomic updates rise even more.

Split Journaling Approach. We therefore explored design alternatives in order to avoid the performance impact of the naive approach. Being the key reason for slow performance, the requirement always to have the on-disk Merkle hash tree in a consistent state needs to be relaxed. In fact, it is desirable to omit updates of Merkle tree nodes for short periods of time, for example, during high load or to minimize latency. In order to have the required hash sums available for post-crash integrity checking nonetheless, they need to be written to an alternative, more efficient data structure. Journaling file systems solve this problem: they allow for efficient and atomic updates of distributed file data and metadata, which in our case includes hash sums that enable integrity checking. A growing journal to which hash sums of updated leaf nodes are appended eliminates the need to immediately update disk blocks that store higher-level tree nodes; they may be flushed from the buffer cache later. This strategy also reduces cryptographic overhead, as only the leaf nodes of the tree are updated frequently.

Protecting the Journal. As the journal now contains information that is critical to ensuring integrity, it must be cryptographically protected, too. To keep the performance benefit, appending records to the journal must not require additional updates of other metadata (e.g., like the root node of the Merkle tree). We ensure journal integrity by continously hashing all appended records. New records are written to the end of the journal together with a new incremental hash sum that authenticates all preceding journal content, thereby enabling incremental integrity checking. To prevent forging of these journal hash sums, we first hash a random secret that is kept in tamper-resistent sealed memory and therefore unknown to an attacker who is trying compute new hashes. Additionally, we encrypt confidential metadata in the journal using AES in CBC mode. This incremental, keyed hashing and encryption scheme is well-understood and, for example, used by Maheshwari et al. in TDB [19].

We will discuss the frequency and granularity of intermediate journal hash sums in Section 3.5, following a detailed discussion of the structure and semantics of jVPFS journal records in Sections 3.3 and 3.4.

3.2 Architecture Overview

We are building on our previous work on VPFS [1] and integrate a consistency mechanism into its architecture. Figure 3 gives an overview of the jVPFS stack. The three top layers only deal with the concept of files, a namespace, and per-file security-critical metadata. They essentially implement a memory file system within the TCB. Another trusted layer called *Sync_manager*, which is located directly underneath this memory file system, implements support for making jVPFS state persistent. Sync_manager is called by the buffer cache whenever data needs to be read into cache buffers or evicted from them. Applications can influence write back through explicit operations such as fsync(), if required.

In order to avoid the complexity of managing a physical storage medium in its own codebase, Sync_manager

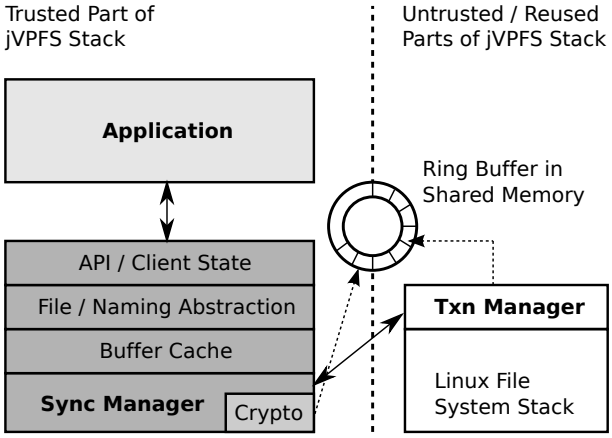


Figure 3: *Detailed view of the jVPFS file-system stack: jVPFS implements a memory-based low-complexity file system within the TCB of the application using it. Through Sync_manager, jVPFS reuses an untrusted Linux file system to make file system contents persistent.*

maps files in the virtual private file system (as seen by the trusted application) to files in an untrusted Linux file system. To distinguish between these two views of a file, we shall refer to the latter ones as *file containers* in the *untrusted storage*.

Cooperation. Like the trusted part of the original VPFS, Sync_manager transparently encrypts and decrypts all data and metadata it exchanges with untrusted components in the storage strack. Furthermore, it calculates and verifies cryptographic hash sums in order to ensure integrity of any data and metadata it receives from untrusted code. In jVPFS, it also performs a minimal amount of state tracking so as to ensure that only consistent changesets are written to persistent storage. Cooperation between Sync_manager and the Linux infrastructure it reuses is enabled by the untrusted *Txn_manager*. It receives from Sync_manager requests and consistency-related hints and translates them into Linux file-system calls. That is, Txn_manager writes cryptographically protected data to file containers and appends records to the journal, which is a file in the Linux file system, too. jVPFS makes extensive use of existing infrastructure, as it exploits any consistency guarantees the underlying Linux file system might provide (e.g., write ordering).

Communication Interface. Trusted and untrusted parts of the jVPFS stack cooperate using a narrow message passing interface and a ring buffer located in a shared memory area. Table 3.2 lists all message types. During normal operation, Sync_manager sends Read_block and Exec_ops messages to request uncached

Message type	Description
Read_block	Read a specific data block
Exec_ops	Execute buffered operations
Write_checkpoint	Create a new journal, which also marks a new checkpoint
Read_checkpoint	Read FS root info from last consistent checkpoint
Read_journal	Read set of complete transactions from journal
Init_shm	Set up shared memory once

Table 1: *Complete list of message types that Sync-manager uses for communication with Txn_manager.*

data blocks, or to flush operations and journal records queued in the ring buffer. Txn_manager handles these requests and writes back encrypted data blocks that are referenced by journal records. Messages of type Read_checkpoint and Read_journal are exchanged at mount time and, if necessary, during recovery. We shall explain the semantics of the Write_checkpoint message in the following section, which covers our approach to journaling and checkpointing of on-disk state.

3.3 Being Prepared for Crashes

Journaling in jVPFS is done at the level of metadata operations. Starting from a consistent set of file containers, which contain the latest *checkpoint* of all file system contents, data blocks are written to untrusted storage and any associated modifications to metadata are logged to the journal. We do not log full blocks of metadata, but only descriptions of specific operations such as updating hash sums or file sizes. Occasionally, Sync_manager flushes all cached blocks—containing both data and metadata—in order to bring all file containers into a consistent state; this state marks a new checkpoint, at which all previously written journal records can be discarded. The general idea for recovery after an unclean shutdown is to replay all journaled operations, iteratively updating metadata structures from the latest checkpoint.

Metadata Dependencies. Operation-level journaling allows for simple tracking of metadata dependencies in Sync_manager, which is important for our objective to minimizing complexity within the TCB. Figure 4 illustrates the dependencies of standard file-system metadata:

Inode: The inode of a file contains the file size in bytes that specifies how much data in the last data block is valid file content. In jVPFS, the inode also stores the root hash of the Merkle subtree that protects the file's contents. The inode itself is stored in the inode

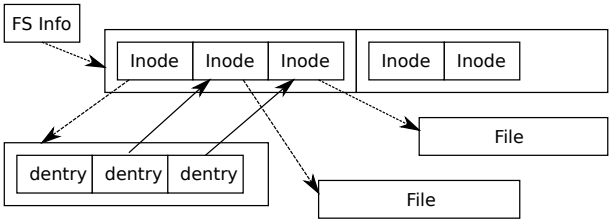


Figure 4: *Dependencies among standard file-system data structures: entries in directories point to inodes, which are associated with files. The superblock-like FS info in jVPFS contains the root hash of the Merkle tree.*

file, which is mapped to its own file container that contains the top levels of the Merkle tree.

Pathname: The inode of a file is referenced by a directory entry, which is stored in a directory file. Directory files form a hierarchical namespace, as directory entries can reference not only regular files, but also other directory files.

The dependencies described above are critical to the consistent representation of a file and must be obeyed by Sync_manager. They translate into the requirement that, for each newly created file, the corresponding inode and the directory entries along the file's pathname need to be written, too. The file and namespace abstraction in the upper layers of the jVPFS TCB already implements most of the required book keeping; a consistent checkpoint will have all this metadata stored in file containers. However, the consistency mechanism that Sync_manager contributes to jVPFS maintains additional state, such that it can log inode and namespace updates to the journal. For each newly created file (including directory files), it keeps the following information in memory:

- A copy of the filename.
- A pointer to the inode of the parent directory.

Sync_manager stores this information in a simple table, which is essentially an extension of the file descriptor table. This volatile *new-file* state is complemented by a one-bit flag in the inode, indicating whether inode and filename of a file have already been written. Sync_manager executes Algorithm 1 to make sure that journal replay restores the inodes and the namespace for newly created files: for a new file and all directories along its pathname, it recursively appends to the journal in reverse path order a File_create record, unless this information has already been logged (or exists in an older checkpoint). Once a File_create record containing a copy of a new file's inode, a pointer to the parent directory's

Algorithm 1 Journaling metadata for new files.

```
function journal_file_create(File *file) {
  // check, if file creation has already been logged
  if (file.inode.is_logged == true)
    return;
  // not announced in journal, get "new file info"
  struct new_file *info = new_file_info_table[file.fd];
  // lookup of file descriptor of parent dir: if it is
  // still open, its creation may need to be logged, too
  int p_fd = info.parent_file_handle;
  int p_iptr = info.parent_inode_ptr;
  File *p_dir = get_file_descriptor(p_fd, p_iptr);
  if (p_dir != NULL) {
    // parent dir still open, make sure it is logged
    journal_create_file(p_dir);
  }
  // announce new file in journal
  file.inode.is_logged = true;
  File_create_rec rec(file.inode, p_iptr, info.name);
  append_to_journal(rec);
}
```

inode, and the filename has been journaled, data blocks can be written to the file container (see next paragraph).

Metadata operations such as rename() or unlink() are logged analogously (i.e., with their parameters). Note that allocation bitmaps or other free-space information need not be considered, because the security-critical part of jVPFS delegates this functionality to the untrusted legacy file system. Also, no explicit write-barrier operations are required, as partial replay of the generated journal records may, at worst, recreate an empty directory or a zero-size file.

Writing User Blocks. The key requirement to be met when writing a block at the leaf level of the Merkle tree is that its hash sum needs to be written to the journal first. Sync_manager prepares write back of user-data blocks; it performs the following operations:

1. Calculate new hash sum over plaintext of updated block's contents.
2. Encrypt block, put ciphertext into free buffer space in shared memory area.
3. Put Block_update record containing updated hash sum and new file size into ring buffer.

Actually writing the data block to persistent storage is done by the untrusted parts of the jVPFS stack. Txn_manager ensures atomicity of block writes by enforcing the following three constraints (note that this scheme is conservative and potentially expensive in

terms of write barriers; we will discuss optimizations for the common case, including write batching, in Section 4):

1. Updated hash sums must reach the journal before the actual block is written to the file container. The underlying legacy file system must be made aware of this write-before relation, for example, by calling fsync() on the journal file.

2. Should a system crash interrupt the write back operations initiated by Txn_manager, the aforementioned order for journal and block writes ensures that either (a) the new hash sum is persistent in the journal and can be used to authenticate the updated block, or (b) the the old version of the block can be authenticated using the old checksum still available in the corresponding on-disk Merkle tree node.

3. Before updating the same block a second time, Txn_manager must make sure that the first update reached stable storage, because of point 2.

It is assumed that the underlying legacy file system can guarantee that aligned writes with a size equal to its own block size are atomic. Most journaling file systems do meet this requirement in their standard configuration using ordered or data journaling [14].

Writing Metadata Blocks. Case 2(b) mentioned above implies that the previous version of a block's hash sum is guaranteed to be available during replay. To meet this requirement at all times, Txn_manager treats metadata blocks differently from blocks with user data. Metadata blocks contain either intermediate nodes of the Merkle tree, or any block from a directory or inode file. Whenever a metadata block is flushed and would overwrite the latest checkpointed version of itself, Txn_manager rescues a copy of the original version into the journal, thereby preserving it in case replay becomes necessary. Consequently, it is not necessary to log hash sum updates of metadata blocks. Sync_manager flags Block_update records as user or metadata, such that Txn_manager can handle the two block types correctly.

Checkpoints. Our split journaling scheme ensures that critical metadata can be restored after a crash. However, letting the journal grow indefinitely would effectively make jVPFS a log-structured file system [22]. This class of file systems requires complicated garbage collection, which in turn would add considerable complexity to the TCB (i.e., Sync_manager). Sync_manager therefore flushes all dirty user and metadata blocks occasionally. Once no more dirty state is in the trusted buffer cache, it signals Txn_manager with a Write_checkpoint message that a new consistent checkpoint can be established. The untrusted Txn_manager then execute the following steps:

1. Process all journal records still queued in ring buffer, submit all block updates to legacy FS.

2. When encountering a special Checkpoint record, instruct legacy FS to make all file containers persistent.

3. Atomically swap current journal file with newly created journal containing just the Checkpoint record.

Note that flushing the buffer cache must be part of the TCB to support any persistency scheme. However, the above checkpointing algorithm does not require any garbage collection in security-critical code, nor does it have to deal with the complexities of writing data to the storage medium safely. It is easy to see how a jVPFS instance can be fully reinstantiated from checkpointed file-system state (in fact, the umount() operation in jVPFS is identical to the checkpoint operation). In the following section, we shall discuss how to restore post-checkpoint state after an unclean shutdown.

3.4 Recovering From Crashes

If the system did indeed crash, the untrusted commodity file system must recover first. Once the untrusted storage has been remounted and the virtualized Linux is booted up, jVPFS can start its own recovery process as we shall now explain.

Mounting a Checkpoint. At mount time, Sync-manager requests from Txn_manager the first record stored in the journal, which is the Checkpoint record containing the FS root info. Sync_manager decrypts the FS root info using the platform's sealed memory implementation and validates its integrity and freshness. Note that the write-back strategies explained in the previous section ensure that this operation succeeds even after an unclean shutdown. However, this assumption may not hold, if a successful attack (see attacker model in Section 2.1), a hardware failure, or a software issue outside the TCB damaged the first journal record. If the Checkpoint record could not be read or validation fails, an integrity error is reported to the application and the file system remains inaccessible. If the journal contains just the Checkpoint record, Txn_manager switches to normal operation mode and behaves as described in Section 3.3. Otherwise it prepares replay.

Preparing Replay. Our operation-level journaling scheme makes the following assumptions:

1. To ensure consistency and integrity, operations specified in journal records must be replayed in the precise order in which they were logged.

2. Journal records encode incremental updates to metadata blocks and modify the previous version of the block, starting with the version that was valid in the last checkpoint.

Requirement 2 dictates that, if there is a checkpointed version of a metadata block preserved in the journal, this version must initially be used during replay—even if a newer version reached its in-place location in the file container. To meet this requirement, Txn_manager copies all metadata blocks it finds in the journal (if any) back to their in-place locations just before replay starts.

Replaying Metadata Operations. Replay is cooperatively performed by both Sync_manager and Txn_manager: the former requests journal records by sending a Read_journal message. Txn_manager responds by filling the ring buffer with a set of records that end with a special record carrying an intermediate keyed hash, which authenticates all preceding journal records (see Section 3.1). Sync_manager then executes the following replay algorithm for each set of journal records provided by the untrusted part:

1. Decrypt all journal records in shared ring buffer, put decrypted versions into private memory buffer, which is inaccessible to untrusted code so as to prevent time-of-check-time-of-use (TOCTOU) attacks.

2. Check integrity of decrypted records using keyed hash sum from last record; in case of mismatch, abort and report integrity error.

3. Re-execute operations specified in all records.

To replay metadata operations such as creating, moving, or unlinking files, Sync_manager reuses existing jVPFS APIs and executes the same code paths that handle calls from an application; the required parameters are extracted from the respective journal records. Handling of filenames is slightly different, as those are recorded relative to their parent directories, which are referenced by their inode number rather than a full pathname.

Handling File Contents. Replaying update records for user data blocks is performed similarly as part of the above algorithm: updated file size information is written to the inode, the hash-sum update is applied to the direct parent node in the Merkle tree. Note that this parent node can always be retrieved and authenticated, because either it was never overwritten or, as a metadata block, it has been preserved in the journal—or an updated version has been generated earlier during replay.

However, since user-block contents are not journaled, file containers always contain the latest version of a block that reached stable storage. On the other hand, the journal may contain multiple Block_update records for the same block. Therefore, Sync_manager skips out-of-date hash sums until it finds the correct record for the user block. It eagerly requests each block during replay, checks its integrity, and applies the hash sum update if it matches the block's contents. A correctly behaving Txn_manager that obeys write-ordering constraints can always provide the latest matching version; misbehavior results in a stale hash sum that will be detected eventually.

We shall evaluate the complexity of jVPFS' consistency mechanism in Sections 5.1 and 6.

3.5 Journal Details

Now that we introduced the various types of journal records, we take a closer look at how the journal is protected in detail.

Confidentiality. The journal contains confidential metadata information such as filenames, so its contents must be encrypted. All payload data of the journal records, including parameters that are passed to internal jVPFS APIs during replay, are encrypted. However, as the untrusted Txn_manager must update file containers in a consistent way based on Sync_manager's constraints, some information cannot be concealed. In particular, we keep the location of data blocks unencrypted, such that untrusted code can write them to their correct locations. Furthermore, we reveal the type of blocks (user or metadata), so as to enable Txn_manager to preserve consistent checkpoints, which are essential for recovery. We consider this an acceptable tradeoff, because an attacker could also learn this information by observing access pattern in the Linux VM.

Integrity. The continuously calculated keyed hash that protects the journal is anchored in the FS root info of the last checkpoint through a random secret stored in it. Thus, a journal is bound to exactly one checkpoint. By embedding intermediate hash sums into the journal, Sync_manager can designate transactions; records between two intermediate hashes can only be authenticated all together, thereby preventing partial replay. We exploit this construction to ensure that security-critical metadata operations described by multiple records are replayed completely or not at all (replay stops in the latter case).

Freshness. Naturally, incrementally calculated hashes cannot reliably mark the end of a data stream (as the HMAC [21] scheme does). As a result, Sync_manager cannot determine from hash sums in the replayed journal, if untrusted components withhold any transactions from the end of the journal; doing so would constitute an attack on freshness. By storing the latest journal hash in tamper-resistent sealed memory before a crash occurs, Sync_manager could detect such an attack during replay: the hash marking the last replayed transaction must match the trustworthy copy preserved in sealed memory. For performance reasons, updates of sealed memory should be done only once for each checkpoint, or an application may request a freshness guarantee explicitely through an fsync()-like operation for transactions between checkpoints.

In our prototype implementation, sealed memory updates are currently dummy operations.

3.6 Managing File Containers

Removal of a file or—in the general case—truncation of it is a metadata operation that jVPFS must log in the journal. However, care must be taken when actually truncating the underlying file container. Assume that recovery becomes necessary after an unclean shutdown. Sync_manager can replay the truncate operation, however, as journal replay always starts relative to a checkpoint, other operations need to be re-executed before it. Some of these operations may depend on file contents that are to be removed, which must therefore still exist for replay to succeed. This is particularly important for metadata files (i.e., the inode file and directories), as logged operations may need to modify them during replay. As a consequence, we must not truncate file containers in the legacy FS right away—even if file truncation has already been logged. Txn_manager therefore builds a list of file truncation requests from truncation records it receives from Sync_manager; once a new checkpoint is persistent, truncated parts of files will finally be obsolete and Txn_manager will garbage collect them in idle time.

4 Optimizations

Intuitively, one would assume that ordered updates of the journal and file containers incur significant performance overhead. However, I/O costs can be reduced drastically by optimizing *untrusted* code.

Write Batching. New journal records and encrypted data blocks are buffered in the shared memory area, until there is no more space or the application explicitly requests a synchronous write (e.g., by calling fsync()). Buffering reduces communication overhead and enables write batching. Batched writes require fewer synchronous writes, because Txn_manager can coalesce a large number of record appends into few journal updates. The benefit is twofold: first, the underlying legacy file system requires fewer I/O operations and at most one write barrier to update the journal file. Second, the legacy file system may write blocks to file containers according to its own optimized strategies, potentially achieving higher performance.

Relaxed Write Order. A write barrier after updating the journal ensures that user blocks can be updated safely. Synchronizing in-place updates of user data blocks that may still be in-flight allows Txn_manager to submit new updates to those same blocks again. However, many common write workloads do not perform any block updates at all (e.g., writing new files or growing them). For these types of workloads, where no old state is modified, the consistency-preserving write-order requirements of jVPFS can be dropped entirely: Txn_manager and the legacy file system may update untrusted storage without enforcing write order, because new data blocks can only be replayed once both their hash sums and the content are persistent; it does not matter which is written first, as long as both are present during replay. Incomplete writes of block–hash sum pairs are treated as if no write operation had been performed at all. In combination with write batching, jVPFS thus achieves I/O overheads close to that of the reused legacy file system, with only few addtional writes to the journal file and occasional checkpointing of Merkle tree nodes.

Note that the functionality for the just described relaxation is implemented almost entirely in untrusted components. Sync_manager only provides a hint indicating whether an older block exists; the hint is trivially computed by checking if the current hash sum in the parent node is null or not.

Out-of-Order Reads. Many write operations can be queued in the ring buffer and it may take a long time to process them. Txn_manager checks if block read requests it receives asynchronously are independent of pending writes; if they are, it handles the read requests immediately without the latency that flushing of pending writes first would cause.

Exploiting Existing Infrastructure. For our evaluation presented in Section 5, we used ReiserFS [23] and NILFS [24] as the underlying file system. With ReiserFS, Txn_manager can only use the POSIX function fsync() to order writes for consistency. This POSIX system call guarantees that all data and metadata of a

file have reached stable storage upon return. However, this persistence guarantee is stricter than what jVPFS requires: in the common case, we just require that certain I/O operations do not overtake each other (e.g., journal records with updated block hash sums are written before modifying the file container or not at all).

The log-structured file system NILFS can ensure a strict order of write operations without calling an explicit API such as fsync(). NILFS ensures that writes reach stable storage in the same order in which an application issued them. Our prototype implementation of Txn_manager exploits this behavior to eliminate I/O delays caused by fsync(), if possible. We extend reuse of existing consistency support even further by leveraging support for efficient checkpointing of file system state that is built into NILFS. Whenever Sync_manager wants to checkpoint its own file system state before starting a new journal, we create a checkpoint in the underlying NILFS file system.

5 Evaluation

We built jVPFS on a platform based on the Fiasco.OC [7] microkernel from the L4 family. The kernel ensures strong isolation of trusted and untrusted components and uses kernel-protected capabilities to enable secure resource access. The trusted part of jVPFS and test applications utilize libraries and services of the L4Re user-level environment [25]. jVPFS hooks into the generic, POSIX-like VFS interface of L4Re. We use L4Linux [26], a paravirtualized Linux 2.6.36 kernel, to run the untrusted parts of the jVPFS stack.

5.1 Complexity

Table 2 shows a breakdown of the source complexity of the jVPFS stack, which is written in C++ (cryptographic library routines are in C). All figures were generated using David A. Wheeler's 'SLOCCount' [27]. In addition to the subsystems listed in the table, jVPFS also reuses an AVL-tree implementation that is part of the TCB of any L4Re application. It comprises approximately 800 lines of C++ code. The L4Re VFS supports file-system plugins and is also linked to any L4Re application.

The main contribution of jVPFS compared to VPFS is its new persistency layer. In our prototype implementation, it comprises 729 source lines of code (SLOC). The functionality that implements journaling and replay of metadata operations requires 325 SLOC, including cryptographic protection as explained in Section 3.5. This is an order of magnitude smaller than in typical monolithic file systems; for example, the journal block device layer (JBD2) for Ext4 comprises almost 5,000 SLOC in Linux 2.6.36. We attribute this significant reduction of complexity to key design decisions in jVPFS: First, the logic to add operation-level journaling is a simple extension of the code that implements write batching using the shared ring buffer. We mainly added additional record types for different operations (e.g., File_create or File_unlink) and consistency state tracking. Second, Sync_manager reuses the same API entry points as the VFS layer to replay operations; parameters for API calls are retrieved from journal records. We implemented less than a dozen SLOC for replay of each type of operation in a switch statement. The remaining 404 SLOC of Sync_manager's current implementation would be required for persistency anyway (e.g., transfer of data blocks, shared memory setup, ring buffer logic).

Subsystem	SLOC
L4Re: VFS	2,303
jVPFS: memory file system	2,444
jVPFS: Sync_manager (persistency)	404
jVPFS: Sync_manager (journal/replay)	**325**
L4Re: libcrypto	667

Table 2: *Source complexity of jVPFS: Sync_manager contributes 729 lines of code to the TCB. Only 325 lines of code are related to journaling and replay.*

The functionality in the TCB could only be reduced this much, because Txn_manager (which is approximately 1,300 SLOC in size) makes extensive reuse of the complex untrusted Linux file system stack.

5.2 Write Performance

Due to space constraints, we focus our performance evaluation on write and metadata-intensive benchmarks and recovery. We did all benchmarks on the same hardware configuration we used for performance evaluation of the original version of VPFS [1]. The evaluation machine has two 2.0 GHz dual-core Opteron processors and 2 GB of DDR RAM. We restricted the hardware resources to one core and 256 MB of physical RAM in all benchmarks. We used two storage mediums, a 80 GB SATA hard disk (Samsung HD080HJ) and a USB flash disk (Buffalo Firestix, 1 GB).

Using strace, we recorded all file-system calls that benchmarking tools executed on Linux. Like we did for VPFS, C++ programs generated from these traces were compiled for L4Re and used to replay all file-system operations on a jVPFS stack; we also ran Linux versions of the trace players on native Linux without any encryption to establish a baseline. Native Linux could use the full 256 MB of RAM, whereas the jVPFS configuration allocated 64 MB of it to its trusted buffer cache that is isolated from L4Linux. Some traces were also used to

Trace	VPFS	jVPFS
PM-1	2.52 s	1.02 s *(0.16 s)*
bonnie++ (encrypted)	32.0 MB/s	38.4 MB/s
bonnie++ (plaintext)	42.0 MB/s	53.1 MB/s

Table 3: *Performance comparison between jVPFS and original version of VPFS using ReiserFS on the hard disk (VPFS figures taken from [1]).*

Trace	Storage	Base	w/o Jrnl	w/ Jrnl
PM-2	ReiserFS HDD	5.11 s *(0.10s)*	6.37 s *(0.14s)*	9.56 s *(0.27s)*
PM-2	NILFS Flash	27.12 s *(0.20s)*	12.36 s *(0.60s)*	13.49 s *(0.54s)*
untar	ReiserFS HDD	1.61 s *(0.06s)*	2.07 s *(0.02s)*	2.14 s *(0.03s)*
untar	NILFS Flash	7.09 s *(0.04s)*	9.65 s *(0.09s)*	9.83 s *(0.13s)*

Table 4: *Execution times and standard deviation for benchmarks of jVPFS with and without journaling enabled, compared against native Linux as a baseline.*

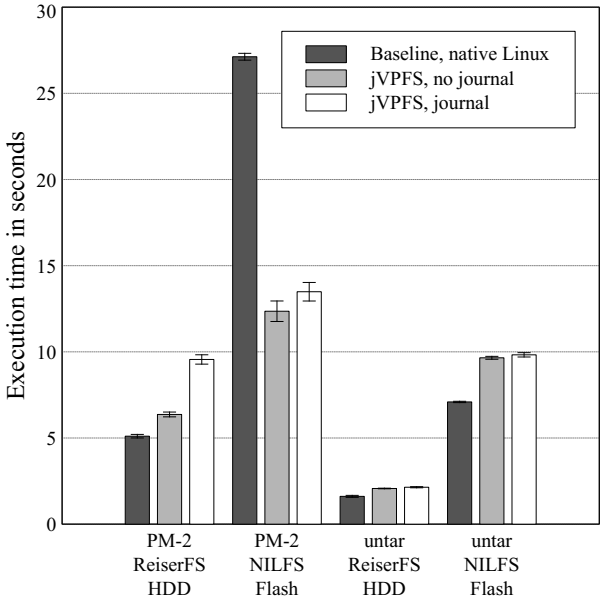


Figure 5: *Benchmark results for Postmark and untar traces; see Table 4 for exact values of execution times with standard deviation.*

benchmark VPFS [1], so we can roughly compare jVPFS against its predecessor, too. Unless stated otherwise, all benchmarks were run ten times and the results averaged; we give standard deviations for increased confidence.

Throughput. We first tested throughput performance by writing two 1 GB files using a bonnie++ trace (see Table 3). With metadata journaling, jVPFS achieves 38.4 and 53.1 MB/s for encrypted and for plaintext files, respectively, with ReiserFS on the hard disk. When we disabled jVPFS journal writes, the underlying legacy file system was eight percent faster to write the unencrypted, but integrity-protected, file containers: effective throughput was 57.1 MB/s, which is close the 58.1 MB/s we measured for native Linux (figures not in Table 3). jVPFS clearly outperforms VPFS (32 and 42 MB/s).

PostMark. PostMark is a synthetic benchmark that creates, modifies, and then deletes a large number of files. The PM-1 trace we used to measure the original VPFS configuration operated on 5,000 files with a size in the order of a few kilobytes [1]. We replayed this mostly-cache workload on jVPFS, which shows significantly better performance than the older VPFS. Another PostMark trace, PM-2 with ten times as many operations on 50.000 files, causes a large number of evictions from the trusted buffer cache and writes to the storage medium. We used this metadata-intensive trace to measure the journaling overhead (see Table 4, or Figure 5 for visual representation). With ReiserFS on the hard disk providing the untrusted storage, we see a 1.5x overhead when journaling is enabled, and about a factor of two compared to the baseline. We expected such a behavior, because calling fsync() on the journal file when required for consistency is expensive on magnetic disks.

With NILFS driving the flash disk, we found that jVPFS can actually perform better than running the benchmark natively in Linux. We determined the strict write ordering of NILFS to be the cause for this unintuitive result: PostMark frequently modifies the small files it created, causing a large number of serialized block updates (i.e., log writes) in NILFS. The jVPFS buffer cache absorbs many of these updates, such that less data actually reach the legacy file system. On the other hand, our system greatly benefits from the ordering guarantees of NILFS, as it does not require synchronous writes to update its journal. Nevertheless, journal writes do cause increased write traffic, as can be seen in the figures. We measured 3.5 MB of journal records, but they arrive in groups smaller than the NILFS block size, thus causing the 9 percent journal I/O overhead we measured.

Untar. The untar trace simulates unpacking a tar archive with thousands of small and large files (kilobytes to megabytes); when done, it flushes all data and metadata to stable storage. We measured 3 and 2 percent journaling overhead for the ReiserFS/HDD and NILFS/flash configuration, respectively. This overhead correlates well with the actual size of the journal file, which ac-

counted for 2.3 percent of all data written to the untrusted storage. These figures are lower than for the PostMark benchmark, as there is a number of very large files among the more than 3,000 files and directories that are created—those files dominate write traffic.

The total overhead of the jVPFS stack over the Linux baseline in this benchmark is 33 percent for ReiserFS and 39 percent for NILFS. Virtualization, increased communication costs, and cryptographic operations contribute to this overhead. While significant, but we believe this overhead is acceptable considering the security advantages jVPFS has over monolithic systems.

5.3 Recovery Performance

We tested jVPFS' recovery functionality using the untar trace both in simulation and on real hardware.

Simulations. In simulation mode, we let Txn_manager terminate itself after it logged a specific amount of data to the journal (about 70 percent of the files were written up to that point). We did not power-cycle the machine, but only restarted Txn_manager and the trusted component of jVPFS. Sync_manager successfully replayed all records from completed transactions as reported by Txn_manager. We then ran a test application that tried to open all files referenced in the journal and read their contents. In total, jVPFS recovered 13 directories containing 1,761 files, which could all be opened and read. Metadata for 26 files was not recovered, because the last transaction they were part of was incomplete; the application received an ENOENT error for these files. This test succeeded reliably and no integrity errors were found.

We repeated the tests with journal writes being disabled, such that jVPFS behaved like the original VPFS. Txn_manager was allowed to write Checkpoint records only. After the simulated crash, the file system could be remounted, but the application received an integrity error when the trusted file-server component of jVPFS tried to look up names in the root directory. The file system was inaccessible afterwards and all data was lost.

Real Hardware. We then tested our system on real hardware. We power-cycled the machine right in the middle of the benchmark and let Linux recover the partition containing the legacy file system. Due to its strict write ordering, NILFS quickly recovered file containers and a valid jVPFS journal, which could successfully be replayed. In multiple tests, hundreds to thousands of files were recovered, depending on the exact moment of the power loss. For example, in one particular instance our system restored 2,710 files consisting of 9,826 blocks of user data within 5.1 seconds; the journal contained 1.2 MB of valid metadata updates. All recovered files could be read; for all other files, the aforementioned test application received an ENOENT error. No corruptions (i.e., intergrity errors) were reported, as we expected.

In the configuration utilizing the hard disk, ReiserFS replayed varying numbers of transactions in its own journal and jVPFS recovered files with no errors other than ENOENT for missing files. We did however also use Ext4 in our experiments and got unexpected results: after recovering the Ext4 partition, jVPFS found a checkpoint record in the journal, but no transactions. We determined an Ext4 optimization called "delayed allocation" to be the reason for this behavior: it may produce zero-sized files after recovery, if the application did not call fsync() on the file descriptor. Due to our own optimization in jVPFS, which we explained in Section 4, Txn_manager did not use fsync() in the untar benchmark, except right after the file system had been created and the Checkpoint record was written. We are currently investigating ways to make jVPFS reliable on Ext4, too.

6 Related Work

We shall now discuss other work that relates to the improvements we made to VPFS [1] in order to securely add robustness against unclean shutdowns.

File System Consistency. jVPFS implements journaling and replay of high-level metadata operations. A similar approach is used by journaling file systems such as Windows NTFS. Others, including Ext3/4 and ReiserFS, instead append complete metadata blocks to their journal in order to log inode, directory, and allocation updates [11]. They implement *full-block journaling*. We considered this approach for jVPFS, but rejected it as we found it to be more complex in our architecture. Journaling metadata blocks requires much more fine-grained dependency tracking within the TCB. Operations such as creating a file modify many metadata structures, which are distributed across multiple blocks in the buffer cache. Furthermore, the fact that more than one inode (or directory entry) is stored in a single metadata block causes additional false dependencies. For example, when writing back metadata for one file, the directory block containing the filename might contain an entry for another file that has recently been created, but whose data or inode has not been written yet. Thus, writing such a directory block actually creates an inconsistency in the on-disk state. To avoid having to increase transaction sizes by including a potentially large number of unrelated files, systems that use full-block journaling implement roll-back mechanisms that temporarily remove incomplete updates from metadata blocks before they are written. We tried to integrate such mechansims into jVPFS, but found them

to add more complexity to the TCB than operation-level journaling as described in Section 3.

Transactional file systems such as ZFS [28] share the problem of false metadata dependencies. They use a copy-on-write approach to prevent inconsistent on-disk state in the first place. Instead of updating data and metadata in-place, they write all modified blocks to free space and then adjust pointers to reference those updated blocks. In conjunction with a hash tree, updates must always propagate to the root. As explained in Section 3.1, the overhead incurred by this approach is significant.

The soft update [10] approach makes sure that a consistent file system can always be restored. The key idea is to apply in-place updates in such an order that only minor inconsistencies occur after a crash. Pointers are guaranteed to be valid, however, old and new metadata (or blocks with user data) may be mixed. This relaxation is inherently incompatible with Merkle tree updates. We therefore did not further consider the soft update approach for solving the robustness problems of VPFS.

The journaling scheme in jVPFS is related to the log-structured approach [22]. What sets our system apart from this type of file systems, is that its consistency mechanism is split into two isolated parts, with complex garbage collection not being part of the TCB.

Untrusted Storage. The logging approach the Trusted Database System (TDB) [19] uses to protect its transaction log is similar to that of jVPFS. It also uses a Merkle tree to ensure integrity. However, jVPFS splits the implementation of journaling and replay into two isolated components using a novel cooperation scheme. jVPFS also reuses existing consistency primitives of an untrusted file system, whereas TDB implements a complete, new database in the TCB.

The protected file system (PFS) [29] unifies journaling and hash logging in a way similar to jVPFS in order to securely use untrusted storage. However, it operates at the level of file-system blocks rather than metadata operations and has a monolithic codebase.

SiRiUS [30] is an example for a network file system that uses untrusted servers. It also stacks onto existing network file systems such as Sun NFS [31] and delegates management of persistent file storage to untrusted infrastructure. However, to the best of our knowledge, SiRiUS does not have an integrated recovery mechanism to ensure consistency of its *metadata freshness files*.

Non-standard Consistency Primitives. Systems such as Featherstitch [32] offer efficient means to applications to specify write-before constraints. The untrusted part of jVPFS can benefit from such expressive consistency primitives in the same way as it benefits from write-order guarantees in NILFS.

7 Conclusions

We built jVPFS, a secure stacked file system that implements post-crash recovery with a minimal trusted computing base (TCB): it requires only 325 lines of C++ code for the security-critical functionality of metadata journaling and recovery, which is an order of magnitude less than widely-used Linux file systems require to provide crash resistence. It reuses an untrusted Linux file system, from which it is strictly isolated through address spaces and virtual-machine boundaries. jVPFS delegates most of the work for managing a physical storage medium to the Linux file system stack, while making extensive use of existing consistency primitives. For example, it can exploit strict write-order guarantees offered by NILFS. Thus, the trusted core of jVPFS can operate at a high abstraction level of metadata operations, greatly reducing the complexity that file-system consistency mechanisms usually contribute to the TCB.

jVPFS outperforms its predecessor VPFS in all benchmarks we did and was shown to be much more robust against unclean shutdowns. It successfully and reliably recovered from temporary damage after power loss. Its strong integrity checks did not detect any corruptions in the recovered secure file system, which was layered on top of ReiserFS on a hard disk, or NILFS, a log-structured Linux file system optimized for flash storage.

8 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd Bryan Ford for their valuable feedback and suggestions for improvement of this paper. Thanks also go to the members of the Operating System Research group at Technische Universität Dresden for helpful discussions and feedback. This work has been supported by the German Research Foundation (DFG-Geschäftszeichen HA 2461/9-1).

References

[1] Carsten Weinhold and Hermann Härtig. VPFS: Building a Virtual Private File System With a Small Trusted Computing Base. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 81–93, New York, NY, USA, 2008. ACM.

[2] The Month of Kernel Bugs (MoKB) Archive. `http://projects.info-pull.com/mokb/`, November 2006.

[3] Apple Inc. About the security content of iPhone OS 3.1.3 and iPhone OS 3.1.3 for iPod touch. `http://support.apple.com/kb/HT4013`, February 2010.

[4] Removing iPhone 3G[s] Passcode and Encryption. http://www.youtube.com/watch?v=5wS3AMbXRLs, July 2009.

[5] Apple Inc. About the security content of iPhone OS 3.0.1. http://support.apple.com/kb/HT3754, August 2009.

[6] iPad Jailbreak - Jailbreak Your iPad. http://www.ipadjailbreak.com/p/jailbreak-your-ipad.html.

[7] The Fiasco Microkernel. Located at: http://os.inf.tu-dresden.de/fiasco/.

[8] Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the Eleventh ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[9] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, 2006.

[10] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: a technique for eliminating most synchronous writes in the fast filesystem. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1999. USENIX Association.

[11] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[12] Valerie Aurora. Soft updates, hard problems. http://lwn.net/Articles/339337, July 2009.

[13] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

[14] Andrea C. Arpaci-Dusseau. Model-based failure analysis of journaling file systems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 802–811, Washington, DC, USA, 2005. IEEE Computer Society.

[15] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling Is Occasionally Correct. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–16, Berkeley, CA, USA, 2008. USENIX Association.

[16] Trusted Computing Group. Trusted Platform Module. http://www.trustedcomputinggroup.org/developers/trusted_platform_module.

[17] ARM Limited. ARM Security Technology - Building a Secure System using TrustZone Technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[18] Matt Blaze. A Cryptographic File System for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.

[19] Umesh Maheshwari, Radek Vingralek, and Bill Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 135–150, San Diego, CA, oct 2000.

[20] R. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[21] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Message Authentication Using Hash Functions: the HMAC Construction. *CryptoBytes*, 2(1):12–15, 1996.

[22] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 1–15, New York, NY, USA, 1991. ACM.

[23] ReiserFS on Namesys website (archived 2007). http://web.archive.org/web/20071023172417/www.namesys.com/, 2007.

[24] NILFS - Continous Snapshotting Filesystem for Linux. http://www.nilfs.org/en/.

[25] Adam Lackorzynski and Alexander Warg. Taming subsystems: capabilities as universal resource access control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, pages 25–30, New York, NY, USA, 2009. ACM.

[26] L4Linux Website. http://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[27] D. Wheeler. *SLOCCount*. available at: http://www.dwheeler.com/sloccount/.

[28] ZFS Website. http://hub.opensolaris.org/bin/view/Community+Group+zfs/WebHome.

[29] C. Stein, J. Howard, and M. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Technical Conference*, pages 79–90, 2001.

[30] E. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *Proceedings of the 10th Network and Distributed Systems Security (NDSS) Symposium*, pages 131–145, February 2003.

[31] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. *Innovations in Internetworking*, pages 379–390, 1988.

[32] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *SOSP '07: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, pages 307–320, New York, NY, USA, 2007. ACM.

Semantics of Caching with SPOCA: A Stateless, Proportional, Optimally-Consistent Addressing Algorithm

Ashish Chawla, Benjamin Reed, Karl Juhnke[†,*], Ghousuddin Syed
{*achawla, breed, gsyed*}*@yahoo-inc.com,* [†]*yangfuli@yahoo.com*
Yahoo! Inc

Abstract

A key measure for the success of a Content Delivery Network is controlling cost of the infrastructure required to serve content to its end users. In this paper, we take a closer look at how Yahoo! efficiently serves millions of videos from its video library. A significant portion of this video library consists of a large number of relatively unpopular user-generated content and a small set of popular videos that changes over time.

Yahoo!'s initial architecture to handle the distribution of videos to Internet clients used shared storage to hold the videos and a hardware load balancer to handle failures and balance the load across the front-end server that did the actual transfers to the clients. The front-end servers used both their memory and hard drives as caches for the content they served. We found that this simple architecture did not use the front-end server caches effectively.

We were able to improve our front-end caching while still being able to tolerate faults, gracefully handle the addition and removal of servers, and take advantage of geographic locality when serving content. We describe our solution, called SPOCA (Stateless, Proportional, Optimally-Consistent Addressing), which reduce disk cache misses from 5% to less than 1%, and increase memory cache hits from 45% to 80% and thereby resulting in the overall cache hits from 95% to 99.6%. Unlike other consistent addressing mechanisms, SPOCA facilitates nearly-optimal load balancing.

1 Introduction

Serving videos is an I/O intensive task. Videos are larger than other media, such as web pages and photos, which not only puts a strain on our network infrastructure, but also requires lots of storage.

Our clients access videos using web browsers. They connect to front-end servers which serve the video content. The front-end servers cache content, but are not the permanent content repository. Videos are stored in a storage farm that is made up of network attached storage accessible by all front-end servers.

To further complicate things, the video content is spread around the world. So, when a client requests content that is non-local, we must decide whether to have the client pull from the remote cluster that has the content, or copy the content from the remote cluster and serve it locally.

Video delivery is fastest and causes the least amount of load on the rest of the infrastructure if the content is cached in the memory of a front-end server. If the content must be accessed from the disk of the front-end server, the load on the front-end server increases slightly. It causes significantly more load and slower delivery if the content must be retrieved from the storage farm. Increased load on the serving infrastructure translates into higher cost to upgrade networking components and to add more servers and disk drives in the storage farm to increase the number of operations per second that it can handle. Thus, good caching at the front-end servers is important to latency, throughput, and the bottom line.

The Yahoo! Video Platform has a library of over 20,000,000 video assets. From this library, end users make about 30,000,000 requests per day for over 800,000 unique videos, which creates a low ratio of total requests to unique requests. Also, because videos are large, a typical front-end server can hold only 500 unique videos in memory and 100,000 unique videos on disk. The low ratio of total/unique requests combined with the large size of video files make it difficult to achieve a high percentage of cache hits.

A straightforward architecture, shown in Figure 1, uses a VIP (Virtual IP) load balancer which distributes requests in a round-robin fashion among a cluster of front-end servers. The VIP exposes an IP address that

[*] This work was done when the author was employed at Yahoo! Inc.

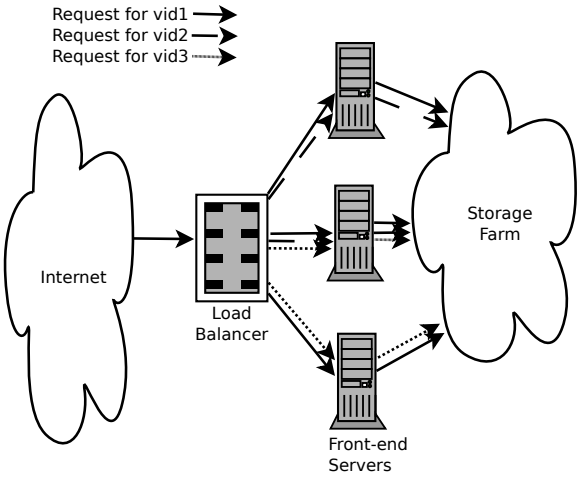


Figure 1: A straightforward content serving architecture using a hardware load balancer VIP (Virtual IP) with front-end server that are connected to a shared storage farm.

the clients connect to. The VIP routes connections to front-end servers to balance load and mask failures of front-end servers. Front-end servers manage their cache by promoting every requested item while demoting the least recently used item.

This was our initial content serving architecture that we used in production. Unfortunately, this straightforward approach results in more requests to the storage farm, due to cache misses at the front-end servers, than the farm can handle. Our storage farm has copious space but limited bandwidth. The front-end server disks are a secondary bottleneck because memory cache misses exhaust the disk throughput before the CPU of the front-end server can be fully utilized.

In a cluster of front-end servers behind a VIP, each piece of popular content will end up cached on multiple servers. For example, in Figure 1 `vid1` is a popular video that ends up cached on all the front-end servers. `vid2` and `vid3` may have only been requested twice each, but that resulted in each video being cached on two nodes.

This is grossly inefficient compared to caching each piece of content on only one server and routing all requests for that content to the server where it is cached. If the cluster's collective cache stores as few copies of each video as possible, then it will be able to store as many unique videos as possible, which in turn will drive down cache misses as low as possible. Eliminating redundant caching of content also reduces the load on the storage farm. An intelligent request-routing policy can produce far more caching efficiency than even a perfect cache promotion policy that must labor under random request routing.

Maximizing caching efficiency via request routing introduces practical challenges. It is difficult to keep a request router's knowledge in synch with the actual cache of each front-end server. Furthermore, even if a request router has a real-time database of cache contents, a database lookup on every user request is a non-trivial latency. Also, most deployments must have more than one request router, which raises the possibility of two different routers independently making a different decision of where to place new content that is not yet in cache, or where to re-locate content when a front-end server leaves or enters the cluster.

The cache promotion algorithm is a natural place to look for improvement. A better promotion policy offers us significantly more memory cache hits and is therefore a step toward relieving the disk throughput bottleneck. The accesses to the storage farm, however, are reduced only marginally by a good promotion policy, because the disk cache miss percentage is low to start with. In some circumstances, delaying a page-in from permanent storage to the disk cache until we are confident that a piece of content is promotion-worthy actually results in more disk cache misses than automatic promotion does. Therefore, another solution is necessary.

Of course the above discussion does not consider the problems arising from the geographic distribution of the content. Not all content is available at all locations. The cluster of servers closest to the user, *nearest locale*, may not be the cluster storing the content, *home locale*. So when we put together the caching discussion and the geolocality, we end up with the following possible user experiences:

1. nearest locale and cached ⇒ excellent experience
2. nearest locale and not cached ⇒ average experience
3. home locale and cached ⇒ average experience
4. home locale and not cached ⇒ below average experience

To get excellent user experience for the most users we need to be able to cache popular remote content locally.

In this paper we describe SPOCA, a system for consistent request routing, and Zebra, a system for routing requests geographically. Both systems have been in production for a few years now at Yahoo! The contributions of this work are:

- We describe a system that is actually used in production in a global scenario for web-scale load.
- We show the real world improvements we saw over the simple off-the-shelf solution in terms of performance, management consolidation, and deployment flexibility.
- SPOCA implements load balancing, fault tolerance, popular content handling, and efficient cache utilization with a single simple mechanism.
- Zebra handles geographic locality in a way that fits

nicely with the mechanism used locally in the data centers.

- We are able to implement all the above with only soft state.

2 Requirements

Our content distribution network is faced with different traffic profiles for its various delivery modes. To handle this, profiles are divided by types of content into pools. This allowed us to adjust the provisioning and policies of the pools to accommodate the traffic profiles. The three main content pools are: Download pools (DLOD), Flash Media Pools(FLOD), Windows Media Pools(WMOD). FLOD is made up of a relatively small library of files and a high average popularity; for DLOD the traffic consists mostly of a large number of unpopular files; and WMOD streaming must deal with both a huge library and some very hot streams. A high level goal for the platform was to merge these pools and be able to manage the diverse requirements of the different traffic profiles in an adaptive way.

A naïve approach to partitioning a pool among a group of front-end servers would be to maintain a catalog that associates each content file with a particular front-end server. The catalog approach is, however, impractical because the set of servers in a location is constantly changing. It would be too time-consuming to re-index the entire content library every time a new front-end server became available, or a current server became unavailable, or a former server re-entered the pool after having been temporarily unavailable. Therefore Yahoo! uses a stateless addressing approach.

For stateless addressing, the inputs are a filename and a list of currently available servers; the output is a server from the list. This eliminates the need to maintain and communicate a catalog. The tradeoff is that Request Router must recalculate the destination server on every request, but fortunately the cost is only microseconds in practice. The same input always produces the same output, so two different Request Router servers, without communicating to each other, will address the same file to the same front-end server within a pool.

We have additional stringent requirements for our addressing algorithm. First, it must partition the set of filenames proportional to different weights for different front-end servers in heterogeneous pools. For example, a newer server might have twice the capacity of an older server, and therefore should serve twice as large a portion of the content library.

The proportionality requirement rules out the use of a distance-based consistent hashing algorithm, although such algorithms are consistent and stateless. Such an algorithm assigns an address to each front-end server, and

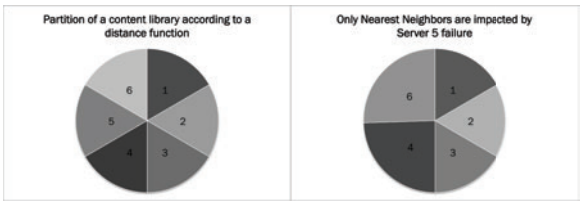


Figure 2: Distance-based Consistent Hashing

assigns a file to the server with the address closest to the hash of the filename, according to some distance function. To see that distance-based consistent hashing does not respect server weights, suppose to the contrary that some configuration of front-end server addresses did respect server weights. Suppose that some server then became unavailable. All of the content for which the unavailable server had been responsible would fall to its nearest neighbors by the distance function. Servers farther away by the distance function would pick up none of the load. Therefore, the content library would no longer be partitioned proportional to server weights (see Figure 2).

In some circumstances it might be reasonable to permit the load of a missing server to be redistributed to its nearest neighbors only. For Yahoo!, however, one reason for a server to be missing is that the server was overloaded. If its load then falls entirely on a few neighbors by distance, they may also become overloaded and fail as well, creating a domino effect that brings down the entire pool. Therefore it is critical that whenever a server leaves the pool, its load is distributed among all remaining servers, proportional to their respective weights. (Figure 3 show an acceptable redistribution.)

Our second requirement for our addressing algorithm is that it be optimally consistent in the following sense: when an front-end server leaves or enters the pool, as few files as possible are re-addressed to different servers, so that caching is disrupted as little as possible.

The two requirements of respecting weight and readdressing a minimum number of files are quite limiting. For example, suppose that a pool has three front-end servers of weight 100, and two servers of weight 200. If a new server of weight 200 is added to the pool, not only must the new server be assigned two-ninths of the files in the content library as a whole, but more specifically for each of the other servers it must take over two-ninths of the files that server was handling.

Figure 3 depict roughly what must happen when servers join and leave the pool if weights are to be respected and a minimum number of files are to be reassigned.

A third requirement on our hashing algorithm arises from the fact that a proportional distribution of files

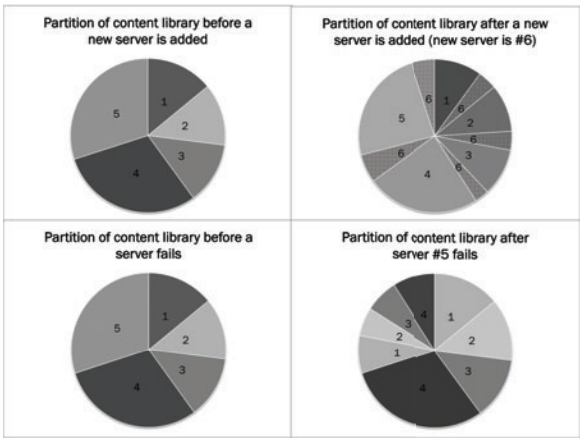


Figure 3: Proportional Redistribution of Load

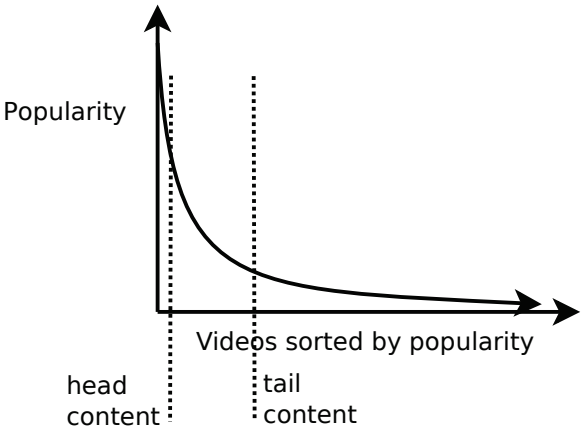


Figure 4: Video popularity follows a power law distribution.

among servers does not necessarily result in a proportional distribution of requests. Perfect caching is in conflict with perfect load balancing, because some files may be more popular than others. Indeed, a single file may be hotter than any front-end server in the pool could handle by itself. There must be a way to detect hot files and/or detect overloaded servers, so that traffic can be distributed away from affected servers.

The load-balancing requirement could be hacked in to the distribution system as an exception to our consistent hashing algorithm, for example simply by saying that files beyond some popularity threshold are evenly distributed between all front-end servers.

Instead of distributing popular streams to all front-end servers, we distribute only to two or three or however many are necessary to meet demand. The means that our algorithm must produce more than a consistent server as output; it must produce a consistent server list.

Figure 4 shows distribution of requests over the content served by our video service. Most requests are for a small number of popular or head content; however, there are still many requests spread over the long tail of less popular content. Similar distributions of popular content have been observed for other services as well [18].

The head content served by Yahoo! is so popular that a single front-end server cannot handle all user requests for a single video. The request router must have a mechanism to distribute hot content to more servers than one front-end server. A few extra cache misses are less problematic than a server being completely overwhelmed by a piece of head content.

The majority of user requests for video, however, are requests that could (and should) be handled by a single server. Therefore, it is equally important that the number of pieces of tail content addressed to each server be proportional to that server's capacity. Although our front-end servers are generally homogeneous, new front-end servers we add to the cluster inevitably have different capacity than existing servers. As we distribute requests across the cluster, we need to take into account the different capacities of the front-end servers.

Our platform is deployed as a cluster of video servers spread across the world, so we need to take geolocality into account. However for unpopular content, it is more effective to serve the content from a remote location rather than try to make the content geographically local. As content becomes more popular we want clients to access them from servers that are close to them.

Finally, our video service is a 24/7/365 operation. We need it to be elastic: we need to be able to add and remove servers from the running system. We also need it to be fault tolerant: we need to gracefully handle the failure and recovery of front-end servers.

3 Overview

We drive caching and locality decisions based on content popularity. Zebra decides which non-local popular content should be cached closer to the requestor. Local content will be cached at an optimal number of local servers based on popularity.

Zebra routes popular requests for popular content to the cluster closest to the nearest locale and unpopular content to the home locale. Zebra initially considers all content as unpopular, so the first request for a particular video will be directed to the home locale. Subsequent requests for the same content will cause Zebra to consider the content as popular and direct requests to the nearest locale. Because of the number of videos served we want to do this popularity detection in a way that uses only soft state and can be tracked with a fixed, and relatively small, amount of memory.

Local content caching uses a Stateless, Proportional, Optimally-Consistent Addressing Algorithm (SPOCA)

to route content requests to our front-end servers rather than simple VIPs. Given the name of a piece of content and a list of front-end servers, the algorithm deterministically maps the content to a single front-end server. Two request routers will independently arrive at the same mapping, and the same request router will make the same mapping repeatedly without having to remember anything. The computation is faster than database lookups, and the only communication overhead required is the list of active front-end servers.

SPOCA is consistent beyond being deterministic. Front-end servers will occasionally drop out of the cluster due to outages or maintenance, and new servers will occasionally be added to refresh technology or increase capacity. When the list of active servers changes, it is unacceptable for the mapping of content to servers to wholly change. Indeed, for optimal consistency, content should never be re-mapped from server A to server B unless A left the cluster or B joined the cluster. In other words, if server A and server B are present both before and after the cluster changes, then no content can be remapped from one to the other.

To serve content we assign each piece of content served a unique name the form `Content-ID.domain`. The `Content-ID` is deterministically derived from the identifier of the content, the hash of the filename for example. `domain` corresponds to the pool to which a piece of content belongs. `domain` also represents a valid DNS subdomain managed by Yahoo!. Thus, `Content-ID.domain` is a valid DNS name that can be resolved by a DNS server. We have a special DNS server, called Polaris, that works with Zebra and SPOCA to route a request to the appropriate server.

During DNS resolution Zebra and SPOCA determine the front-end server to route the request to; Polaris directs the client to that server by resolving the request for `Content-ID.domain` to the determined front-end server's IP. Web pages that embed content to be served uses a URL of the form `http://Content-ID.domain` to address the content.

4 The Zebra Algorithm

Zebra handles the geographic component of request routing and content caching. Its main caching task is to decision out when requests should be routed to content's home locale and when the content should be cached in the nearest locale. Zebra makes this decision based on content popularity.

Zebra tracks popularity using soft state with a limited amount of memory. Bloom filters [2] seem like a good data structure to use for this kind of tracking. As requests for content come in, we can add them to the bloom filter to track popularity. Unfortunately, it is not possible to remove content from the bloom filter. So we need a way to stop tracking content that is no longer popular.

Rather than using a single bloom filter, Zebra uses a sequence of bloom filters (we use 17 filters in production). Each bloom filter represents requests for a given interval, on the order of hours. We keep a fixed number of filters in the sequence and expire older filters as new are added. Content is considered popular based on the union of the intervals. We optimize the popularity check by combining all the bloom filters older than the current interval into one after the start of a new interval, so that popularity checks involve lookups in only two bloom filters.

Note that even if we had used a more sophisticated bloom filter structure, such as counting bloom filters [7], we would still need to track entries to be deleted because they are no longer popular. Our strategy of using a sequence of bloom filters both tracks and removes entries that are no longer popular using simple bloom filters.

If content is deemed popular, it is in one of the two bloom filters, the content will be cached locally. Not only does Zebra enable more effective geographic caching, it also enabled us to decouple delivery from storage. It now makes sense to have content serving front-ends in a data center that has no content storage servers. Popular content will be cached locally at the front-end servers and unpopular content requests will be routed to its home locale.

Zebra determines which serving clusters will handle a given request based on geolocality and popularity. SPOCA determines which front-end server within that cluster will cache and serve the request.

5 The SPOCA Algorithm

To maximize the cache utilization at the front-end servers and thereby minimize the load on our storage farm SPOCA aims to localize requests for a given video at a single server. This will allow the best utilization of the aggregate memory of the front-end servers. We also need to balance the load across the front-end servers and handle failures and server additions. Requests for a popular video may overload a single server, so we need to be able to assign the handling of such content to multiple front-end servers. Finally, we have serving clusters around the world, so we need to take geolocality into account.

Figure 5 illustrates how we would like content to be served. Each video is served by one server. This decreases the load on the storage farm and it more effectively uses the cache of the front-end servers. Since `vid1` is a popular video we would like it to be served by multiple servers.

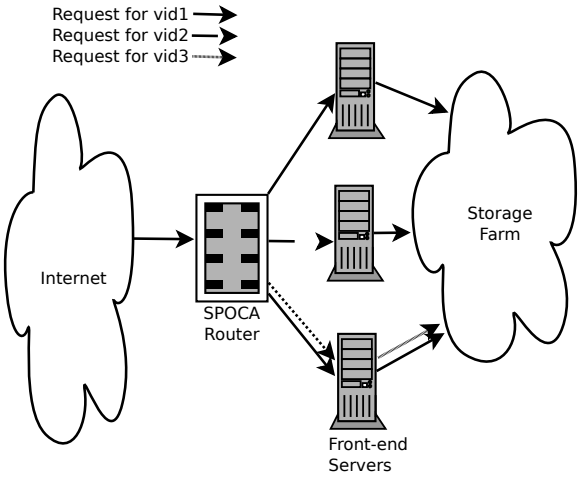


Figure 5: SPOCA consistently routes the same video to the same front-end server. It also increases the number of servers serving popular content. (e.g. `vid1`)

SPOCA fulfills our requirements using a simple content to server assignment function based on a sparse hash space. Each front-end server is assigned a portion of the hash space according to its capacity.

The SPOCA routing function takes as input the name of the requested content and outputs the server that will handle the request. The SPOCA routing function uses a hash function to map names to a point in a hash space as shown in Figure 6. Each front-end server is assigned a portion of hash space proportional to its capacity. Not every point in the hash space maps to a front-end server, so when the hash of the name of a requested video maps to unassigned space, the result of the hash function is hashed again until the result lands in an assigned portion of the hash space.

Using this hashing scheme SPOCA load balances content requests using random load balancing in such a way that it can gracefully handle failures, the addition and removal of front-end servers, and popular content.

5.1 Failure handling

If a front-end server fails, the portion of the hash space that was assigned to the failed server becomes unassigned as shown in Figure 7. Requests that would have been assigned to the failed server are rehashed as normal until they land in a region assigned to an active server. This has the nice property that only the content assigned to the failed server will be re-routed to other servers in a balanced fashion. Content assigned to the servers that have not failed will continue to be served by those servers, which allows us to continue to utilize effectively the cached content at those servers.

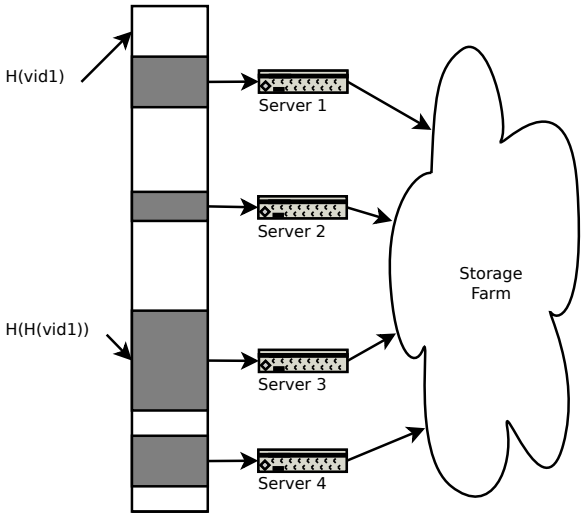


Figure 6: An example assignment of the SPOCA hash map. The name of the requested video initially hashes to empty space, but when hashed again is assigned to server 3.

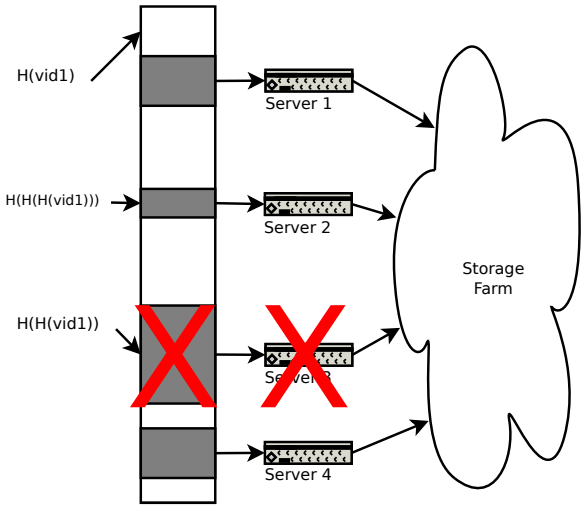


Figure 7: When server 3 fails, the content handling for the named video is reassigned to server 2.

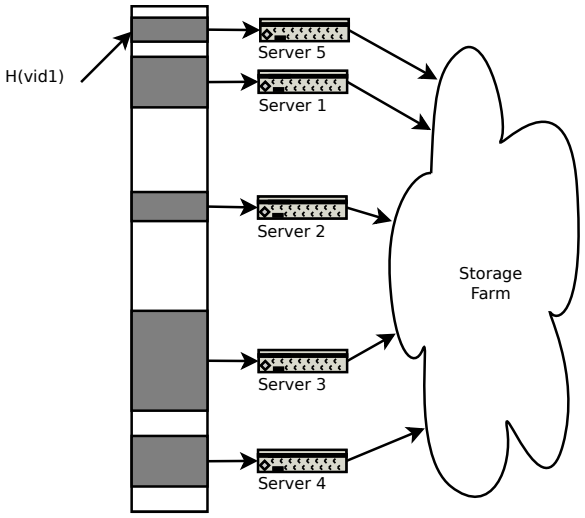


Figure 8: When server 5 is added, the content handling for the named video is reassigned to server 5.

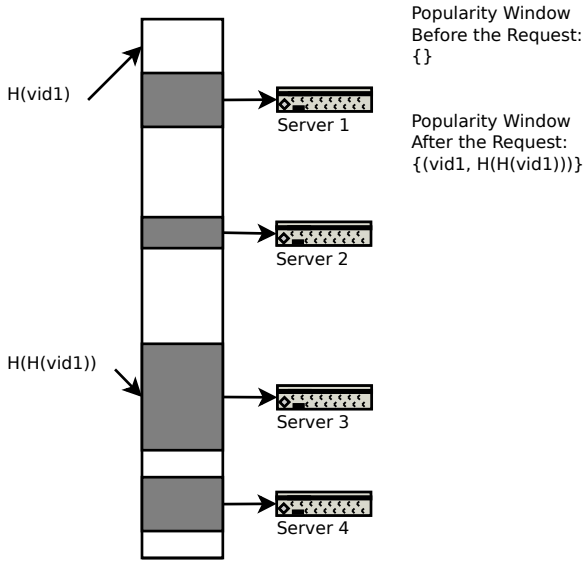


Figure 9: When a request for `vid1` is received, SPOCA routes the request to server 3 and stores the hash in the popularity window.

5.2 Elasticity

New servers are mapped into the unassigned portion of the hash space as shown in Figure 8. When this happens a portion of the content assigned to other servers will now be assigned to the new server. For example, in Figure 8 the video that was previously handled by server 3 will now be handled by the new server, server 5. Server 3 may have `vid1` content in its cache because of previous requests. Eventually server 3 may end up replacing `vid1` with other content it is serving. If `vid1` becomes popular or server 5 fails, server 3 will again start serving `vid1`.

Servers are removed from service by simply removing their assignments from the hash space. This will cause the mechanism described in the previous section to kick in and content served by the removed server will be spread to other active servers.

5.3 Popular content

SPOCA tries to minimize the number of servers that cache a particular piece of content to maximize the aggregate number of cached objects across the front-end servers. This strategy works well with tail content, unpopular content, but it can cause front-end servers for head content, popular content, to become overloaded. So, for head content we need to route requests to multiple front-end servers. We do this routing using a simple extension to the SPOCA routing mechanism described earlier. Popular files are handled by the same algorithm that deals with missing servers, so a page-in for either may benefit the other.

To handle popular content, the request router stores the hashed address of any requested content for a brief popularity window, 150 seconds in our case. On every new request, the request router checks whether it has a saved hash for the requested content. If no hash is present, the request is routed to a front-end server using the normal procedure. If a hash is present, meaning the content has been served within the popularity window, routing will start using the stored hash rather than the name of the requested content. In either case, only the final hash (i.e. the address where the request was ultimately routed) is saved along with the content name.

Figure 9 shows the routing for `vid1` with the popularity window. Because popularity window did not have a entry for `vid1` the request will be routed the same as Figure 6. If another request is received in the popularity window, as shown in Figure 10, the routing will start with the hash stored in the popularity window rather than the hash of `vid1`. Note that the server that handles overflow is the same server as handles requests for `vid1` if server 3 fails as shown in Figure 7.

When the popularity window expires, the stored hash for each object is discarded regardless of how recently it was used. It follows that each object may be mapped to as many different servers as there are requests for that file within the popularity window. For a given object, the sequence of servers to which it is routed is the same in each popularity window, with the number of requests determining how deeply into the sequence the mapping advances. An object which is never requested twice within the same popularity window is always mapped to the same server.

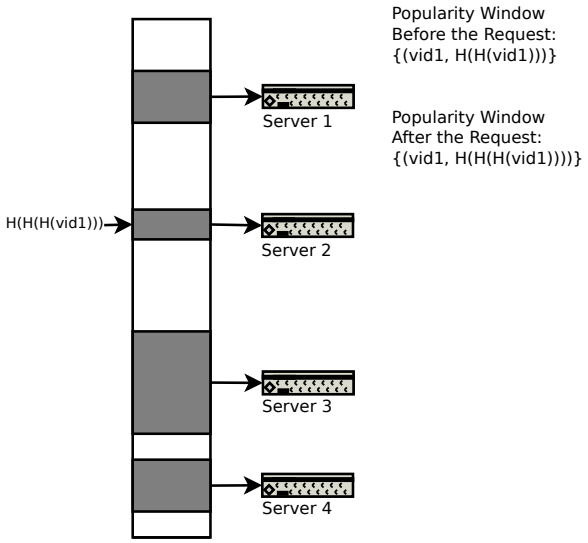


Figure 10: When the next request for `vid1` is received, SPOCA uses the hash in the popularity window to start the routing and routes to server 2 and stores the updated hash in the popularity window.

If a file is temporarily popular enough to be distributed to two servers, both will cache it. If later the first server is unavailable, the second server will resume primary responsibility for the file it already has cached, rather than that file being re-assigned elsewhere. Similarly if a server goes down for a while, the files for which it is responsible will be cached elsewhere. If, after that server has come back on line, one of its files becomes popular enough to be distributed to a second server, that second server will already have the file in cache.

The load balancing for hot files is not absolutely stateless, but the request router does not need to store an entry for each of the 20,000,000 unique files in the library, or even the 800,000 unique files requested in a day, merely the 20,000 unique files it has seen in the last 150 seconds. This amount of data can easily be held in memory.

The larger the time window, the greater the permitted load imbalance between servers. The shorter the time window, the greater the unnecessary duplication of files. It is a balance between inefficiencies.

Keeping a histogram of recent files would be greater overhead, and would in any case not answer the important question, namely whether we prefer the inefficiency of load imbalance or the inefficiency of redundant copies. The information necessary to decide about that tradeoff is not available to the stream router.

5.4 Memory management

The original media server caching policy waited too long to bring content from the filers into local cache. We decided to use a more aggressive policy with SPOCA. When explaining this policy we use the terms page-in and page-out to describe the actions of populating the cache with a piece of content and evicting a piece of content from the cache. The new caching policy, embodied in SPOCA, goes to the extreme to correct the problem of unresponsiveness: SPOCA calls for content to be paged into local cache as soon as it is requested. This raises an obvious question: is immediate page-in the right approach?

The traditional concern with aggressive caching is that it causes churn, which is to say that a less-popular item will be brought into cache, forcing a more-popular item to be deleted from cache. The old caching system reflected the traditional mindset: that system was designed to prevent churn by tracking whether an item was truly popular before paging it in. In our configuration, however, churn is not the primary issue.

In light of increasing disk sizes and improvements to the distribution of requests among media servers, it is reasonable to suppose that a media server can cache every stream that is requested for an entire week. We observed that whenever a file is requested, the probability that it will be requested again within a week is 69% for ads, 80% for audio, and 83% for video, so even a random new file we are getting our first request for is likely to be more popular than the oldest file in our cache.

However, even if churn is low, even if cache misses have been reduced to an absolute minimum, there is another potential reason not to page in aggressively, namely that paging in itself causes load on the filer. If we recall that the objective of the caching system is to reduce filer load, then we must reduce both cache misses and page-ins. The policy of immediate page-in may save less in cache misses than it costs in page-ins.

Indeed, an average page-in may place a greater burden on a filer than an average cache miss, because in the case of a cache miss, the user may not view the entire stream, so the filer can quit serving it partway. We have a question that cannot be decided in the abstract: It takes real data to know whether 20% of the average stream is viewed, or more, or less.

It is possible, however, that a page-in places a lesser load on a filer than a cache miss. This is because the filer can serve a page-in request at full speed, reading the whole file contiguously, whereas to stream a file the filer has to stream it out byte by byte.

What we observed was that for the video media servers, the immediate page-in policy is correct. In most cases we load the filer the least by paging in immediately, which works very well with our general desire to be as responsive as possible.

5.5 SPOCA Implementation

SPOCA's consistent hashing algorithm implementation is based on the standard C pseudo-random number generator. In extreme circumstances, linear congruence generators may have undesirable properties, but for distributing traffic based on filename they are quite sufficient.

Since each file has a `Content-ID` this number is used as the *seed* for our pseudo-random number generator. Thus it can generate an arbitrarily-long, deterministic sequence of numbers, uniformly scattered in the unit interval.

When an front-end server is entered into a pool, it is assigned a segment of the unit interval that does not overlap with any other server's segment. The length of the segment represents that server's weight. Upon receiving a request for a file, SPOCA generates pseudo-random numbers within the unit interval until one lies within a segment that has been mapped to a server. Algorithm 1 shows the basic logic to map a request for content, `filename`, to a server. At the heart of the algorithm is the function `maptoserver(seed)`, which returns the server whose assigned segment includes `seed`. Failures are also reflected in `maptoserver(seed)`. If SPOCA detects that a front-end server has failed, `maptoserver(seed)` will return `null` for any `seed` that falls in the failed servers assigned interval.

When a new server is added to the pool, the load is evenly distributed among all the servers. What it means is that the new server takes some load from each of the existing servers in the pool. On the same lines, when a server goes down, it takes the load which that server was handling and re-distributes to the other servers in the pool.

Algorithm 1 Pseudo-Random Generation Algorithm

```
1: seed := filename;
2: seed := rand(seed);
3: while maptoserver(seed) = NULL do
4:     seed := rand(seed);
5: end while
6: return maptoserver(seed)
```

In order to allow for the painless addition of future front-end servers, our servers typically cover only 1% to 25% of the unit interval, depending on our anticipation of future growth. Note that if the mapping of front-end servers covers only 1% of the unit interval, then SPOCA will have to generate an average of 100 pseudo-random numbers to distribute one request. Even so this algorithm causes negligible latency, because linear congruence generators are so simple.

On the rare occasion that one wishes to add servers after an interval becomes entirely mapped, it is best to shrink all existing segments to a some fraction of their size, causing a corresponding fraction of disruptions in caching. However, sparsely assigning the unit interval, creates enough room for years of foreseeable needs. By starting with only 1% of the unit interval assigned, we can grow a cluster to almost 100 times its initial size before worrying about running out of room. In our experience so far, we have been able to plan ahead and avoid any such re-mapping.

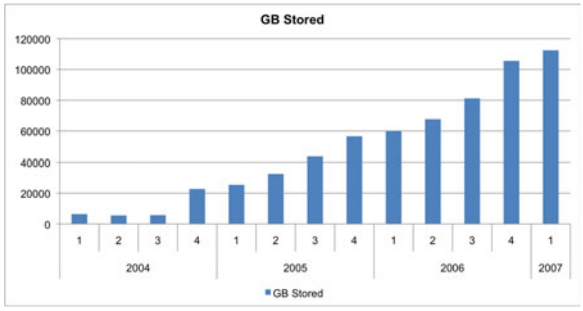


Figure 11: Data Growth

SPOCA controls distribution of popular files by saving a `seed` for each `filename` for a configurable length of time T. Every time T elapses, all saved seeds are thrown away. If a request arrives for a file for which the request router has a saved `seed`, that file is deemed to be popular, and the generation of pseudo-random numbers starts from the saved `seed` rather than from the `filename`. Algorithm 2 shows how the previous algorithm can be modified to incorporate this behavior.

Algorithm 2 Distribution Algorithm

```
1: if savedseed[filename] = NULL then
2:     seed := filename;
3: else
4:     seed := savedseed[filename];
5: end if
6: while maptoserver(seed) = NULL do
7:     seed := rand(seed);
8: end while
9: savedseed[filename] := seed
10: return maptoserver(seed)
```

A stream requested N times within the interval T may be distributed to as many as N servers. Every time the saved seeds are thrown away, the pseudo-random sequence starts over from the filename. If a file is never popular enough to be requested twice within T, the filename will always be used as the seed, and thus it will always be distributed to the same server.

6 Evaluation

We use historical data we have collected over time from production to evaluate the quality and performance of our proposed algorithms. The dataset shown in Figure 11 covers Q1 2004 to half of Q1 2007. The amount of content stored and distributed has been approximately doubling year-over-year.

When storage and distribution each double, requests served from the filers quadruple. We could run into a situation where we would need four times the filers to support scaling delivery by two times. The reason for this is because the cost model was not linear. There were limitations on IO performance of the filers and it was not a linear growth. We were consistently seeing 100% CPU hit on the filers even for the 10% cache miss. Without SPOCA, we would need more hardware as we start serving more requests from the filer.

We observed in our production environment that load balancing with SPOCA is three times better than load balancing by VIP because SPOCA's hashing function deterministically routes to the right server to serve the request whereas VIP routing does simple random routing without regard to where content may be cached. Note that this is not one server getting three times the load of another, as might happen in a peer-to-peer system which does not guarantee a partition of the address space proportional to server weights. The variation among servers is rather three times the small amount produced by random request routing. This increased variation is smoothed out by the law of large numbers: the more requests distributed by the request router, the closer to an average load each front-end server gets. For the delivery platform, load balancing has never been an issue since SPOCA was implemented, whereas the caching problem SPOCA solved was quite serious.

Over 99% of files accessed in any given day are not accessed often enough to trip the popularity trigger. Therefore these files are cached on only one front-end server each. As a file grows more popular it does not necessarily get distributed to the entire cluster. Instead it is spread to two, three, four servers and so on in linear proportion to how popular the file is. No more cache misses are created than are necessary for load balancing.

Only 0.01% of all files are popular enough to be cached on all front-end servers. No additional mechanism for identifying such files is necessary. Each server's probability of being the next server in the sequence to which a hot stream is mapped is proportional to that server's weight. The fail over mechanism therefore load-balances extremely popular content in the same proportions as it maps tail content.

With SPOCA we have been able to reduce cache misses by 5x. Each item is now cached on as few media servers as possible (usually one media server in each location instead of all servers in a pool). There is also an increase in stability with larger globally distributed clusters of front-end servers. Our serving clusters grew from 8 servers to 90. The management of this single large cluster was much easier than managing many small clusters. Because SPOCA automatically adapted to different workload profiles on a per file basis, we no longer needed to use separate pools for the different workloads. We were eventually able to consolidate 11 pools into a single pool, which also allowed us to further simplify management.

Without implementing SPOCA, video streaming from filers was approximately 219 terabytes daily. Refer to Figure 12 which shows the graphs for filer storage v/s filer streaming of video assets. After implementing

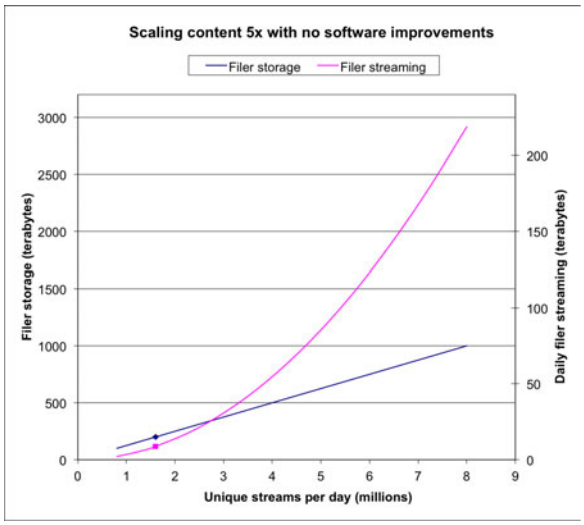


Figure 12: Without Software Improvements

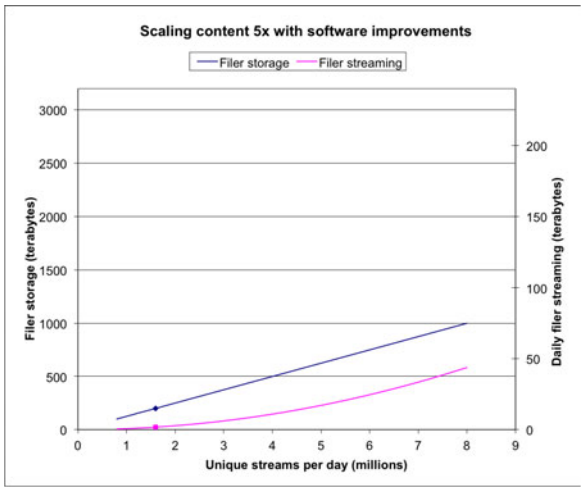


Figure 13: With Software Improvements

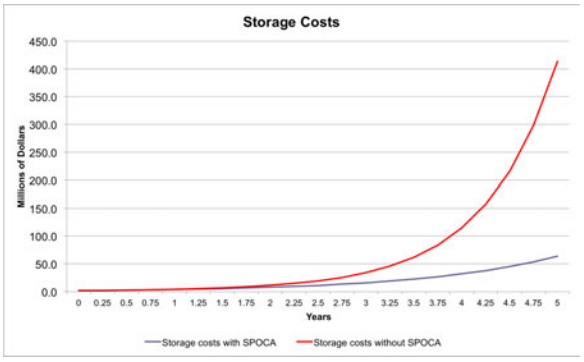


Figure 14: Storage Costs With and Without SPOCA

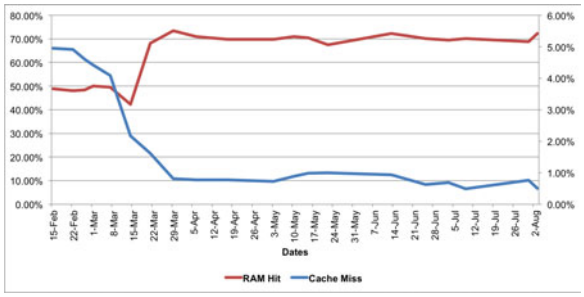


Figure 15: Increased memory cache hits across all server by lowering the popularity window from 300 secs to 240 secs. The change was rolled out to servers over a period of weeks.

SPOCA, we see a drastic reduction (5x) in the number of bytes streamed from the filers. Video Streaming from filers with SPOCA reduced to approximately 44 terabytes daily. Fig 13 shows the performance improvements with SPOCA.

A further benefit of SPOCA has been substantially increased memory cache hits. Torso content (less popular than head content but more popular than tail content) was formerly distributed between many servers. It was in disk cache everywhere, but qualified for promotion to the memory cache nowhere. With consistent addressing, however, torso content requests are collected and focused on a single front-end server, sometimes making the content popular enough for promotion to memory cache on one server before it needs to be paged into disk cache anywhere else.

As the video assets stay more in cache, streaming from filers is reduced and hence the need to add more filers also goes down as we saw from Fig 13.

Fig 14 is a projection model of the costs saving on the filer. More than $350 million dollars in unnecessary equipment (filer costs, rack space etc) alone can be saved in five-year period of running with SPOCA. In addition to the substantial savings in filer costs, the hidden cost savings included Power savings for running the equipments and data center utilizations.

The size of the popularity window is a tunable parameter. A smaller window results in fewer disk cache misses and more memory cache hits at the cost of greater load imbalance due to hot files. The window of 150 seconds for the SPOCA has driven the cache misses low enough while keeping the load balancing is even enough, such that we have not had to fiddle with the parameter to find the perfect sweet spot.

Table 1 shows us the traffic pattern for one of our data centers (S1S). It is pretty impressive that S1S missed 0.7% on the Flash workload and 0.4% on the Download workload on 3/14. The numbers would have been even better, but we had a server that lost it's RAM drive which adversely affected those numbers. The traffic pattern is a little variable, but on the whole SPOCA has apparently reduced cache misses in S1S by almost a factor of ten. From this table (Table 1), we also see that the Download RAM hit went down from 70% on 3/7 to 21% on 3/10 and to 14.2% on 3/14. This drop can be attributed to server misconfigurations or the in-memory index database getting corrupted after server reboots.

In production we measured how drastically we reduced the costs to store the bytes on the memory v/s the disk cache. Our measurements indicate that SPOCA improved memory cache hits from from 45% to 70%, and the overall cache hits increased from 95% to 99.6%. Due to this an item stays in cache for an increased duration (5X of the time it stayed earlier. e.g. what stayed in for only three hours now stays for 15 hours). We can also scale storage hardware linearly instead of quadratically, thus directly positively impacting cost.

Fig 15 shows the impact of increased memory cache hits from lowering the popularity window (referred to as T in Section 5.5) from 300 seconds to 240 seconds. This window governs the reset of the sequence of servers which are obtained from `maptoserver(seed)` function. A shorter window results more the cache hits because the requests are concentrated on fewer servers. But there is a trade-off here. If we make the window too small, we can use too few servers to serve a popular stream and overload the servers. The figure shows the three main pools in our three main locations. Further adjustments and measures have lead us to use a popularity window of 150 seconds in our production environment.

The improvement in memory cache hits from SPOCA was less dramatic: it only improved from 49% to 70%. However, the improvement in cache misses was more dramatic: it went from 5% all the way down to 0.5%, i.e. a factor of ten! We later bumped up the memory cache hits by adding more RAM, refreshing some of the older hardware with beefier boxes.

	2/26	3/1	3/5	3/7	3/10	3/14
Download cache miss	9.7%	7.2%	4.3%	3.7%	1.8%	0.4%
Download cache hit	90.3%	92.8%	95.7%	96.3%	98.2%	99.6%
Download RAM hit	42.4%	66.0%	63.4%	70.0%	21.0%	14.2%
Flash Cache miss	21.8%	13.5%	22.0%	14.8%	2.5%	0.7%
Flash RAM hit	57.2%	81.4%	66.1%	71.5%	90.0%	90.1%

Table 1: Cache Hit and Misses for the Download and Flash Pools in S1S data center

6.1 Churn Times

To amplify the hit rate, some classical caching schemes can be employed. For example, prefetching, whereby the content is cached to anticipate future requests. These techniques can also reduce the average hops between servers for content delivery. However, we cannot prefetch all the content, because of various limitations including bandwidth and storage costs. One way to evaluate the effectiveness of a cache is to look at its churn time. We define churn time as the period of time an item remains in the cache. A high churn time means that on average an item stays in cache for a long time before being replaced. There have been various models for studying the right cache size for the content type and churn times [14, 22].

We examined the churn times on our various pools of servers across a couple of different data centers. Table 2 has the statistics on churn times from disk cache and Table 3 has the churn times from memory cache from the various pools across multiple data centers. The way to read the table is this: If something is being removed from cache to make room, then it has not been accessed in X time. In table 2, we see that the churn time for DAL - DLOD pool is 8.2 days, which means that the content stays in disk cache for 8.2 days before its churned out. Similarly, in S1S data center, content would stay in Windows Media Pool (WMOD) for 2 years before its removed. However, if we look at table 3, we see that the memory churn time for DAL - DLOD pool is 2.5 hours and in S1S, churn time for Windows Media Pool is 3.8 hours.

The bigger numbers are based on our projection model because when we did this study, the system was not in production for long enough.

7 Related Work

Figure 16 shows a comparison with options engineering had during the design of SPOCA. Our main requirements at that time were proportional distribution of head and tail content, Consistent addressing/Good caching scheme and the ability to scale by adding/removing servers. Some of the schemes we looked at addressed part of our requirements but none addressed all. We

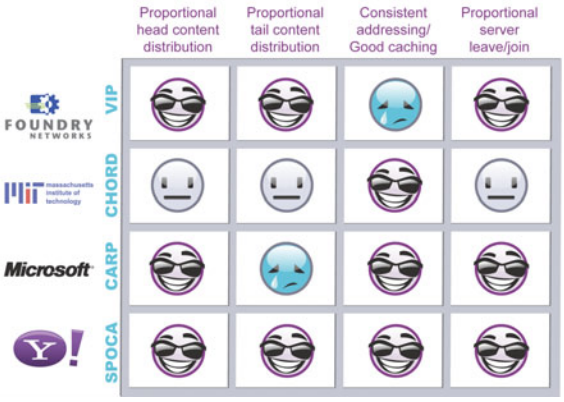


Figure 16: Algorithm Comparisons

had Foundry Networks [3] address three requirements but didn't provided a consistent addressing or caching scheme. Similarly, Microsoft's cache array routing protocol (CARP) [21] partitions the URL space among proxies. CARP uses hash-based routing to provide a deterministic request resolution path and eliminates the duplication of contents that otherwise occurs on an array of proxy servers. CARP made it possible to plug additional servers into the cache array or subtract a server from the array without massive reconfigurations and without significant cache reassigning. Its cache-management features provide both load balancing and fault tolerance. But it failed to deliver proportional tail content and missed on addressing one of our requirement.

Our VIP routers do load balancing based on round robin servicing of connections. Network Dispatcher [11] was early work on this type of router. Round robin DNS is another way to do load balancing. These methods as well as other common methods are described in ONE-IP [5]. As we noted earlier these approaches balance the load, but do not use the aggregate memory of the cluster efficiently.

Caching services such as CoralCDN [8] and Akamai [15] use DNS resolvers to direct clients to caching proxies that are close to clients. Like SPOCA, they serve content to unmodified clients and are excellent at distributing popular content. Much of SPOCA's traffic is made up of a many requests for various unpopular con-

E: (Cache)	DAL	A2S	S1S
DLOD	8.2 days	4.5 days	40 days
WMOD	40 days	5.5 months	2 years
FLOD	9 months	1.4 years	1.5 years

Table 2: Statistics about churn times from the disk cache.

R: (RAM)	DAL	A2S	S1S
DLOD	2.5 hours	35 mins	40 mins
WMOD	1 hour	4.5 hours	3.8 hours
FLOD	4 hours	5.8 hours	6.4 hours

Table 3: Statistics about churn times from the memory cache.

tent that can pollute the front-end caches. For this reason SPOCA does not always choose to direct clients to local front-end servers for unpopular content.

The SPOCA router tries to use the aggregate memory of the front-end servers as one big cache. Locality-aware request distribution (LARD) [16] combined cooperative caching with request routing to achieve load balancing and effective cache utilization. LARD routes content based on load and uses a table indexed by the content identifier to consistently route requests. SPOCA's consistent routing function achieves load balancing without using a table entry for every cached object. We also handle popular content and failures with this same routing function.

As in other consistent addressing schemes [12, 23, 20], we assign each front-end server a section of an address space, and hash file names into this space in order to map files to servers. However, unlike any other scheme we are aware of, the address space is not completely assigned. Some addresses belong to no server. Specifically, a server is not responsible for all addresses between its own address and the address of whichever server is next in the hash space, as in distributed networks such as Chord [19]. Instead, each server is assigned a fixed section of the address space which is proportional to its capacity. Some systems [6, 1] which use consistent addressing mitigate the load balancing problems of Chord by mapping to many virtual servers using a hash function and then mapping sets of virtual servers to physical servers according to load. While this does allow coarse grained load balancing, it still does not handle popular content that needs to be served by multiple machines.

RUSH [10], scheme allows for cluster weights, thus insuring proportionality, but is not optimally consistent. When a cluster is removed or has its weight decrease, it not only causes the necessary shifting from itself to other clusters, it can cause shifting between other clusters. Also dynamic handling of hot streams is not covered.

Peer-to-peer networks are the most common applications of consistent addressing, but in peer-to-peer networks an appropriate partition of the address space is quite difficult to achieve. For example, in a 2005 technical paper, Giakkoupis and Hadzilacos [9] present a method of insuring that the largest section of the partitioned address space is no more than four times the size of the smallest section. Considering the complication that not all servers have equal capacity, their guarantee worsens to an eightfold imbalance between the most-loaded and least-loaded server, relative to each server's capacity.

Moreover, to achieve this factor-of-eight guarantee, Giakkoupis and Hadzilacos weaken the optimal consistency criterion, allowing up to twice the minimum content re-mapping when servers leave the cluster. This further underscores the tension between consistent addressing and proportional load balancing.

Consistent hashing [12] was proposed to handle popular content without swamping a single server. It extends work done by Harvest Cache [4] and Plaxton and Rajaraman [17]. The consistent hashing work was incorporated into a web cache [13] that used consistent hashing to route content requests to servers. Unlike SPOCA the web cache work does not use the same mechanism for load balancing, fault tolerance, and popularity handling. They also always serve requests close to clients, even if the content is unpopular.

8 Conclusion

Zebra and SPOCA routing simultaneously handles our requirements for load balancing, fault tolerance, elasticity, popularity, and geolocality. They do so using a simple mechanisms that nicely handle the different requirements in a consistent way.

Zebra and SPOCA do not have any hard state to maintain or per object meta-data. This allows our implementation not to have to worry about maintaining and recovering persistent state. It also eliminates any per object storage overhead or management, which simplifies oper-

ations.

Operations were also simplified by the ability to consolidate content serving into a single pool of servers that can handle files from a variety of different workloads. Further the ability to decouple the serving and caching of content from the storage of that content allowed us greater flexibility in architecting our caching and storage infrastructure. Specifically, this decoupling allowed us to have front-end clusters without any local content that only cache popular remote content.

In a for-profit corporation, cost savings and end user satisfaction are key success metrics. SPOCA excels on both accounts. We have seen great cost savings with a corresponding increase in the performance of our serving cluster.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *NSDI*, pages 1–16. USENIX Association, 2010.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] Brocade. http://www.brocade.com/index.page.

[4] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *USENIX*, 1996.

[5] O. P. Damani, P. E. Chung, Y. Huang, C. M. R. Kintala, and Y.-M. Wang. One-ip: Techniques for hosting a service on a cluster of machines. *Computer Networks*, 29(8-13):1019–1027, 1997.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazons highly available key-value store. In *SOSP '07: 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007. ACM Press.

[7] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8:281–293, June 2000.

[8] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with coral. In *NSDI'04: 1st conference on Symposium on Networked Systems Design and Implementation*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[9] G. Giakkoupis and V. Hadzilacos. A scheme for load balancing in heterogenous distributed hash tables. In M. K. Aguilera and J. Aspnes, editors, *PODC*, pages 302–311. ACM, 2005.

[10] R. J. Honicky and E. L. Miller. Rush: Balanced, decentralized distribution for replicated data in scalable storage clusters.

[11] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network dispatcher: A connection router for scalable internet services. *Computer Networks*, 30(1-7):347–357, 1998.

[12] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[13] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web caching with consistent hashing. In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1203–1213, New York, NY, USA, 1999. Elsevier North-Holland, Inc.

[14] P. Linga. A churn-resistant peer-to-peer web caching system. In *ACM Workshop on Survivable and SelfRegenerative Systems*, pages 9–22, 2003.

[15] E. Nygren, R. K. Sitaraman, and J. Sun. The akamai network: a platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, 2010.

[16] V. S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. *SIGPLAN Not.*, 33(11):205–216, 1998.

[17] C. G. Plaxton and R. Rajaraman. Fast fault-tolerant concurrent access to shared objects. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 570, Washington, DC, USA, 1996. IEEE Computer Society.

[18] M. Saxena, U. Sharan, and S. Fahmy. Analyzing video services in web 2.0: a global perspective. In *NOSSDAV '08: International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 39–44, New York, NY, USA, 2008. ACM.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM.

[20] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Trans. Netw.*, 6(1):1–14, 1998.

[21] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.1. Internet draft, 1998.

[22] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 1999.

[23] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *SC '06: ACM/IEEE conference on Supercomputing*, page 122, New York, NY, USA, 2006. ACM.

TidyFS: A Simple and Small Distributed File System

Dennis Fetterly
Microsoft Research, Silicon Valley
fetterly@microsoft.com

Maya Haridasan
Microsoft Research, Silicon Valley
mayah@microsoft.com

Michael Isard
Microsoft Research, Silicon Valley
misard@microsoft.com

Swaminathan Sundararaman
University of Wisconsin, Madison
swami@cs.wisc.edu

Abstract

This paper describes TidyFS, a simple and small distributed file system that provides the abstractions necessary for data parallel computations on clusters. In recent years there has been an explosion of interest in computing using clusters of commodity, shared nothing computers. Frequently the primary I/O workload for such clusters is generated by a distributed execution engine such as MapReduce, Hadoop or Dryad, and is high-throughput, sequential, and read-mostly. Other large-scale distributed file systems have emerged to meet these workloads, notably the Google File System (GFS) and the Hadoop Distributed File System (HDFS). TidyFS differs from these earlier systems mostly by being simpler. The system avoids complex replication protocols and read/write code paths by exploiting properties of the workload such as the absence of concurrent writes to a file by multiple clients, and the existence of end-to-end fault tolerance in the execution engine. We describe the design of TidyFS and report some of our experiences operating the system over the past year for a community of a few dozen users. We note some advantages that stem from the system's simplicity and also enumerate lessons learned from our design choices that point out areas for future development.

1 Introduction

Shared-nothing compute clusters are a popular platform for scalable data-intensive computing. It is possible to achieve very high aggregate I/O throughput and total data storage volume at moderate cost on such a cluster by attaching commodity disk drives to each cluster computer.

In recent years data-intensive programs for shared-nothing clusters have typically used a data-parallel framework such as MapReduce [9], Hadoop [13], Dryad [16], or one of the higher level abstractions layered on top of them such as PIG [22], HIVE [14], or DryadLINQ [28]. In order to achieve scalability and fault-tolerance while remaining relatively simple, these computing frameworks adopt similar strategies for I/O workloads, including the following properties.

- Data are stored in streams which are striped across the cluster computers, so that a single stream is the logical concatenation of parts stored on the local file-systems of the compute machines.
- Computations are typically parallelized so that each part of a distributed stream is streamed as sequential reads by a single process, and where possible that process is executed on a computer that stores that part.
- In order to get high write bandwidth on commodity hard drives I/O is made sequential as far as possible. In order to simplify fault-tolerance and reduce communication the computing frameworks do not implement fine-grain transactions across processes. Consequently, modifications to datasets are made by replacing an entire dataset with a new copy rather than making in-place updates to an existing stored copy.
- In order to achieve high write bandwidth the streams are written in parallel by many processes at once. In order to reduce communication and simplify lock management however, each part is typically written sequentially by a single writer. After all the parts have been output in their entirety they are assembled to form a stream.
- In order to provide fault-tolerance when computers or disks become unavailable, the frameworks automatically re-execute sub-computations to regenerate missing subsets of output datasets.

Distributed file systems have been developed to support this style of write-once, high-throughput, parallel streaming data access. These include the I/O subsystem

in River [4], the Google File System (GFS) [11], and the Hadoop Distributed File System (HDFS) [6, 24]. Unsurprisingly there are similarities in the designs of these systems: metadata for the entire file system is centralized and stored separately from stream data, which is striped across the regular compute nodes. They differ in their level of complexity and their support for general filesystem operations: for example GFS allows updates in the middle of existing streams, and concurrent appends by multiple writers, while the HDFS community has struggled with the tradeoff between the utility and complexity of even single-writer append operations [10] to a stream that can be concurrently read. In order to provide fault-tolerance all the systems replicate parts of a distributed stream, and provide reliable write semantics so that a stream append is replicated before the write is acknowledged to the client.

This paper presents TidyFS, a distributed file system which is specifically targeted *only* to workloads that satisfy the properties itemized above. The goal is to simplify the system as far as possible by exploiting this restricted workload. Parts of a distributed stream are invisible to readers until fully written and commited, and subsequently immutable, which eliminates substantial semantic and implementation complexity of GFS and HDFS with appends. Replication is lazy, relying on the end-to-end fault tolerance of the computing platform to recover from data lost before replication is complete, which allows us to eliminate custom I/O APIs so parts are read and written directly using the underlying compute node file system interface.

Sections 2 and 3 outline the data model and architecture of TidyFS in more detail. Section 4 describes some of our experiences operating TidyFS for over a year. We describe related work in Section 5 and then conclude with a discussion of some of the tradeoffs of our design choices.

2 TidyFS usage

This section describes the TidyFS data model and the typical usage patterns adopted by clients. As noted in the introduction, the design aims to make TidyFS as simple as possible by exploiting properties of its target workload. Wherever features are introduced in the following discussion that might seem to go beyond the simplest necessary functionality, we attempt to justify them with examples of their use.

2.1 Data Model

TidyFS makes a hard distinction between data and metadata. Data are stored as blobs on the compute nodes of the cluster and these blobs are immutable once written. Metadata describe how data blobs are combined to form larger datasets, and may also contain semantic information about the data being stored, such as their type. Metadata are stored in a centralized reliable component, and are in general mutable.

TidyFS exposes data to clients using a stream abstraction. A stream is a sequence of parts, and a part is the atomic unit of data understood by the system. Each part is in general replicated on multiple cluster computers to provide fault-tolerance. A part may be a single file accessed using a traditional file system interface or it may be a collection of files with a more complex type—for example, TidyFS supports SQL database parts which are pairs of files corresponding to a database and its log. The operations required by TidyFS are common across multiple native file systems and databases so this design is not limited to Windows-based systems.

The sequence of parts in a stream can be modified, and parts can be removed from or added to a stream; these operations allow the incremental construction of streams such as long-lived log files. A part can be a member of multiple streams, which allows the creation of a snapshot or clone of a particular stream, or a subset of a stream's parts. Clients can explicitly discover the number, sizes and locations of the parts in a stream, and use this information for example to optimize placement of computations close to their input data.

Each stream is endowed with a (possibly infinite) lease. Leases can be extended indefinitely, however, if a lease expires the corresponding stream is deleted. Typically a client will maintain a short lease on output streams until they are completely written so that partial outputs are garbage-collected in the case of client failure. When a stream is deleted any parts that it contained which are not contained in any other stream are also scheduled for deletion.

Each part and stream is decorated with a set of metadata represented as a key-value store. Metadata include for example the length and fingerprint of a part, and the name, total length and fingerprint of a stream. Rabin fingerprints [7] are used so that the stream fingerprint can be computed using the part fingerprints and lengths without needing to consult the actual data. Applications may also store arbitrary named blobs in the metadata, and these are used for example to describe the compression or partitioning scheme used when generating a stream, or the types of records contained in the stream.

2.2 Client access patterns

A client may read data contained in a TidyFS stream by fetching the sequence of part ids that comprise the stream, and then requesting a path to directly access the data associated with a particular part id. This path describes a read-only file or files in the local file system of a clus-

ter computer, and native interfaces (e.g., NTFS or SQL Server) are used to open and read the file. In the case of a remote file, a CIFS path is returned by the metadata server. The metadata server uses its knowledge of the cluster network topology to provide the path of the part replica that is closest to the requesting process. The metadata server prioritizes local replicas, then replicas stored on a computer within the same rack, and finally replicas stored on a computer in another rack.

To write data, a client first decides which stream the data will be contained in, creating a new empty stream if necessary. The client then asks TidyFS to "pre-allocate" a set of part ids associated with that stream. When a client process wishes to write data, it selects one of these pre-allocated part ids and asks TidyFS for a write path for that part. Typically the write path is located on the computer that the client process is running on, assuming that computer has space available. The client then uses native interfaces to write data to that path. When it has finished writing the client closes the file and adds the new part to the stream, supplying its size and fingerprint. At this point the data in this part is visible to other clients, and immutable. If a stream is deleted, for example due to lease expiration, its pre-allocated part ids are retired and will not be allocated to subsequent writers.

Optionally the client may request multiple write paths and write the data on multiple computers so that the part is eagerly replicated before being committed, otherwise the system is responsible for replicating it lazily. The byte-oriented interface of the TidyFS client library, which is used for data ingress and egress, provides the option for each write to be simultaneously written to multiple replicas.

The decision to allow clients to read and write data using native interfaces is a major difference between TidyFS and systems such as GFS and HDFS. Native data access has several advantages:

- It allows applications to perform I/O using whatever access patterns and compression schemes are most suitable, e.g., sequential or random reads of a flat file, or database queries on a SQL database part.
- It simplifies legacy applications that benefit from operating on files in a traditional file system.
- It avoids an extra layer of indirection through TidyFS interfaces, guaranteeing that clients can achieve the maximum available I/O performance of the native system.
- It allows TidyFS to exploit native access-control mechanisms by simply setting the appropriate ACLs on parts, since client processes operate on behalf of authenticated users.
- It gives clients precise control over the size and contents of a part so clients can, for example, write streams with arbitrary partitioning schemes. Pre-partitioning of streams can lead to substantial efficiencies in subsequent computations such as database Joins that combine two streams partitioned using the same key.

The major disadvantage would appear to be a "loss of control" on the part of the file system designer over how a client may access the data, however our experience is that this, while terrifying to many file system designers, is not in practice a substantial issue for the workloads that we target. The major simplification that we exploit is that data are immutable once written and invisible to readers until committed. The file system therefore does not need to mediate between concurrent writers or order read/write conflicts. The detection of corruption is also simplified because data fingerprints are stored by TidyFS. A malicious client is unable to do more than commit corrupted data, or (access-controls permitting) delete or corrupt existing data. In both these cases the corruption will be discovered eventually when the fingerprint mismatch is detected and the data will be recovered from another replica or discarded if no good replicas are available. This is no worse than any other file system: if a client has write access to a file it can be expected to be able to destroy the data in that file.

Systems such as HDFS and GFS perform eager replication in their client libraries. Although the TidyFS client library provides optional eager replication for data ingress, TidyFS gains simplicity and performance in the common case by making lazy replication the default. The potential loss of data from lazy replication is justifiable because of the underlying fault tolerance of the client computational model: a failure of lazy replication can be treated just like a failure of the computation that produced the original part, and re-run accordingly. This is even true for workloads such as log processing, which are often implemented as a batch process with the input being staged before loading into a distributed file system. We believe this reliance on end-to-end fault tolerance is a better choice than implementing fault tolerance at multiple layers as long as failures are uncommon: we optimize the common case and in exchange require more work in error cases.

A drawback to giving clients control over part sizes is that some may end up much larger than others, which can complicate replication and rebalancing algorithms. Both GFS and HDFS try to split streams into parts of moderate sizes, e.g., around 100 MBytes. In principle a system that controls part boundaries could split or merge parts to improve efficiency, and this is not supported by TidyFS since each part is opaque to the system. In our experience, however, such rebalancing decisions are best made with semantic knowledge of the contents of the stream, and

we can (and do) write programs using our distributed computational infrastructure to defragment streams with many small parts as necessary.

The biggest potential benefit, given our current workloads, that we can see from interposing TidyFS interfaces for I/O would come if *all* disk accesses on the compute nodes were mediated by TidyFS. This would potentially allow better performance using load scheduling, and would simplify the allocation of disk-space quotas to prevent clients from denying service to other cluster users by writing arbitrary sized files. We discuss some pros and cons of this direction in the final section.

Of course, the major tradeoff we make from the simplicity of TidyFS is a lack of generality. Clients of GFS use multi-writer appends and other features missing in TidyFS to implement services that would be very inefficient on our system. Again, we address this tradeoff in the Discussion.

2.3 SQL database parts

As mentioned in section 2.1, we have implemented support for two types of TidyFS parts: NTFS files and SQL databases. In the case of SQL parts, each part is a Microsoft SQL server database, consisting of both the database file and the database log file. The TidyFS metadata server stores the type of each part, and this information is used by the node service so that it will replicate both files associated with the part. The HadoopDB evaluation [1] shows that for some data-warehouse applications it is possible to achieve substantial performance gains by storing records in a database rather than relying on flat files. We believe that the ease of supporting SQL parts in TidyFS, compared with the additional mechanisms required to retrofit HadoopDB's storage to HDFS, provides support for our design choice to adopt native interfaces for reading and writing parts. As a bonus we achieve automatic replication of read-only database parts. There remains the problem of targeting these partitioned databases from client code, however, DryadLINQ [28] can leverage .NET's LINQ-to-SQL provider to operate in a hybrid mode shipping computation to the database where possible, as described in the referenced paper.

3 System architecture

The TidyFS storage system is composed of three components: a metadata server; a node service that performs housekeeping tasks running on each cluster computer that stores data; and the TidyFS Explorer, a graphical user interface which allows users to view the state of the system. The current implementation of the metadata server is 9,700 lines of C++ code, the client library is 5,000 lines of mixed C# and C++ code, the node service is 948 lines of C# code, and the TidyFS Explorer is 1,800 lines of C#. Figure 1 presents a diagram of the system architecture, along with a sample cluster configuration and stream. Cluster computers that are used for TidyFS storage are referred to in the following text as "storage computers."

3.1 Metadata server

The metadata server is the most complex component in the system and is responsible for storing the mapping of stream names to sequences of parts, the per-stream replication factor, the location of each part replica, and the state of each storage computer in the system, among other information. Due to its central role, the reliability of the overall system is closely coupled to the reliability of the metadata server. As a result, we implemented the metadata server as a replicated component. We leverage the Autopilot Replicated State Library [15] to replicate the metadata and operations on that metadata using the Paxos [18] algorithm. Following the design of systems such as GFS, there is no explicit directory tree maintained as part of the file system. The names of the streams in the system, which are URIs, create an implied directory tree based on the arcs in their paths. When a stream is created in the system, any missing directory entries are implicitly created. Once the last stream in a directory is removed, that directory is automatically removed. If the parent directory of that directory is now empty, it is also removed, and the process continues recursively up the directory hierarchy until a non-empty directory is encountered.

The metadata server tracks the state of all of the storage computers currently in the system. For each computer the server maintains the computer's state, the amount of free storage space available on that computer, the list of parts stored on that computer, and the list of parts pending replication to that computer. Each computer can be in one of four states: `ReadWrite`, the common state, `ReadOnly`, `Distress`, or `Unavailable`. When a computer transitions between these states, action is taken on either the list of pending replicas, the list of parts stored on that computer, or both. If a computer transitions from `ReadWrite` to `ReadOnly`, its pending replicas are reassigned to other computers that are in the `ReadWrite` state. If a computer transitions to the `Distress` state, then all parts, including any which are pending, are reassigned to other computers that are in the `ReadWrite` state. The `Unavailable` state is similar to the `Distress` state, however in the `Distress` state, parts may be read from the distressed computer while creating additional replicas, while in the `Unavailable` state they cannot. The `Distress` state is used for a computer that is going to be removed from the system, e.g., for planned re-imaging, or for a computer whose disk is

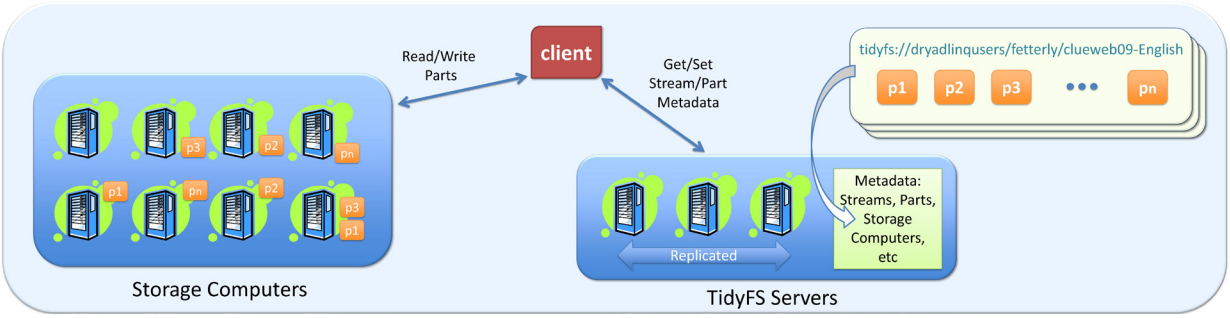


Figure 1: TidyFS System Architecture

showing signs of imminent failure. The `Unavailable` state signifies that TidyFS should not use the computer at all.

In the current TidyFS implementation computers transition between states as a result of an administrator's command. We have found this manual intervention to be an acceptable burden for the clusters of a few hundred computers that we have been operating. A more widely deployed system should be able to automatically detect failure conditions and perform transitions out of the `ReadWrite` state without operator action. This could be implemented using well-known techniques such as the watchdogs employed in the Autopilot system [15].

The metadata server also maintains per-stream and per-part attributes. There is a set of "distinguished" attributes for each stream or part which are maintained by the system automatically or as a side-effect of system API calls, and users may add arbitrary additional attributes as key-value pairs to store client-specific information. For streams, the distinguished values are creation time, last use time, content fingerprint, replication factor, lease time, and length. For parts, the distinguished values are size and fingerprint.

Clients of the metadata server, including the other TidyFS components, communicate with the server via a client library. This client library includes RSL code that determines which metadata server replica to contact and will fail over in case of a server fault.

3.2 Node service

In any distributed file system there is a set of maintenance tasks that must be carried out on a routine basis. We implemented the routine maintenance tasks as a Windows service that runs continuously on each storage computer in the cluster. Each of the maintenance tasks is implemented as a function that is invoked at configurable time intervals. The simplest of these tasks is the periodic reporting, to the metadata server, of the amount of free space on the storage computer disk drives. The other tasks are garbage collection, part replication, and part validation, which are described in the following paragraphs.

Due to the separation of metadata and data in TidyFS and similar systems, there are many operations that are initially carried out on the metadata server that trigger actions on the storage computers in the system. The deletion of streams, via either user action or lease expiration, is one such operation. Once all of the streams that reference a particular part are deleted, every replica of that part should be removed from the storage computer that holds it. The metadata server is responsible for ensuring that there are sufficient replicas of each part, as calculated from the maximum replication factor of all streams the part belongs to. Once the replicas have been assigned to particular computers, the node services are responsible for actually replicating the parts.

In order to determine what parts should be stored on a storage computer, each node service periodically contacts the metadata server to get two lists of parts: the first is the list of parts that the server believes should be stored on the computer; and the second is the list of parts that should be replicated onto the computer but have not yet been copied.

When the node service processes the list of parts that the metadata server believes should be stored on the computer, the list may differ from the actual list of parts on the disk in two cases. The first is that the metadata server believes a part should be there but it is not. This case is always an error, and will cause the node service to inform the metadata server of the discrepancy, which in turn will cause the metadata server to set in motion the creation of new replicas of the part if necessary. The second is that the metadata service believes a part that is present on the disk should not be there. In this (more common) case the part id is appended to a list of candidates for deletion. Once the entire list is processed, the list of deletion candidates is sent to the metadata server, which filters the list and returns a list of part ids approved for deletion. The node service then deletes the files corresponding to the filtered list of part ids. The

complete function pseudocode is listed in Algorithm 1, where `ListPartsAtNode`, `RemovePartReplica`, and `FilterPendingDeletionList` are all calls that contact the metadata server.

The reason for this two phase deletion protocol is to prevent parts that are in the process of being written from being deleted. The metadata server is aware of the part ids that have been allocated to a stream but not yet committed, as outlined in Section 2.2, however these pending part ids are not included in the list of part ids stored on any storage computer.

Algorithm 1 Garbage collection function

```
partIds = ListPartsAtNode();
filenames = ListFilesInDataDir();
List pdList;
for all file in filenames do
    id = GetPartIdFromFileName(file);
    if !partids.Remove(id) then
        pdList.Add(id);
    end if
end for
for all partId in partids do
    RemovePartReplica(partId);
end for
partIdsToDelete = FilterPendingDeletionList(pdList);
for all partId in partIdsToDelete do
    DeletePart(partId);
end for
```

If the list of parts that should be replicated to the node but are not present is non-empty, the node service contacts the metadata server for each listed part id to obtain the paths to read from and write to for replicating the part. Once the part has been replicated the fingerprint of the part is validated to ensure it was correctly copied, and the node service informs the metadata server that it has successfully replicated the part, after which the part will be present in the list of ids that the metadata believes are stored at that computer.

As we will show in Section 4, there is a substantial fraction of parts that are not frequently read. Latent sector errors are a concern for the designers of any reliable data storage system [5]. These errors are undetected errors where the data in a disk sector gets corrupted and will be unable to be read. If this undetected error were to happen in conjunction with computer failures, the system could experience data loss of some parts. As a result, the node service periodically reads each part replica and validates that its fingerprint matches the stored fingerprint at the metadata server; if not, the node service informs the metadata server that the part is no longer available on that computer, potentially triggering a re-replication.

3.3 TidyFS Explorer

The two TidyFS components already described deal with the correct operation of the system. The final component is the graphical user interface for the distributed file system, named the TidyFS Explorer. It is the primary mechanism for users and administrators to interact with the system. Like all other TidyFS clients, the TidyFS Explorer communicates with the metadata server via the client library. For users, TidyFS Explorer provides a visualization of the directory hierarchy implied by the streams in the system. In addition to the directory hierarchy, the TidyFS Explorer exposes the sequence of parts that comprise a stream, along with relevant information about those parts. Users can use the GUI to delete streams, rename streams, manipulate the sequence of parts in a stream, as well as copy parts between streams. Cluster administrators can use the TidyFS Explorer to monitor the state of computers in the system, including determining what computers are healthy, what replications are pending, and how much storage space is available. Administrators can also manually change the state of computers in the system and interact with the node service.

3.4 Replica Placement

When choosing where to place replicas for each part, we would like the system to optimize two separate criteria. First, it is desirable for the replicas of the parts in a particular stream to be spread across the available computers as widely as possible, which allows many computers to perform local disk reads in parallel when processing that stream. Second, storage space used should be roughly balanced across computers. Figure 2 shows a histogram of part sizes in a cluster running TidyFS. Due to this non-uniform distribution of part sizes, assigning part to replicas is not as simple as assigning roughly equal numbers of parts to each computer.

The location of the data for the first copy of a part is determined by the identity and state of the computer that is writing the part. We would like to bias writes to be local as often as possible, so we simply use the local computer if it is "legal" to write there, meaning that the computer is a storage computer in the `ReadWrite` state that has enough space available. We do not know the length of the part before the data is written, so we make a conservative estimate based on typical usage. If the disk fills up, the writer will fail and the computational framework's fault-tolerance will reschedule the computation elsewhere.

Subsequent replication of parts should optimize the two criteria above. We have avoided complex balancing algorithms that directly optimize our desired criterion in favor of simpler greedy algorithms.

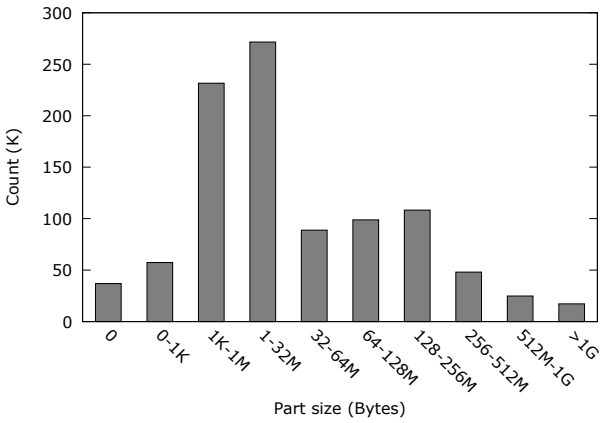


Figure 2: Histogram of part sizes (in MB)

We initially implemented a policy that assigns a replica to the legal computer in the system with most free space. Unfortunately many streams contain very small parts, and after adding one such small part to a given computer it often continues to be the one with the most free space. Since many parts from a stream are replicated in a short time period when the stream is created, this problem resulted in very poor balance for streams with small parts: one computer would hold the second copy of many or all of that stream's parts.

This lack of balance led us to implement a second policy, similar to the one used in the Kinesis project [19] and techniques described in [21]. This approach uses the part identifier to seed a pseudo-random number generator, then chooses three legal computers using numbers drawn from this pseudo-random number generator and selects the computer with the most free space from this set as the destination for the part. As we report in Section 4.4, this second policy results in acceptable balance for streams.

Over time the system may become unbalanced with some computers storing substantially more data than others. We therefore implemented a rebalancing command. This simply picks parts at random from overloaded computers and uses the replication policy to select alternate locations for those parts. The metadata server schedules the replication to the new location, and when the node service reports that the copy has been made, the metadata server schedules deletion from the overloaded computer.

3.5 Watchdog

We recently started to prototype a set of watchdog services to automatically detect issues and report them to an administrator for manual correction. The issues fall into two categories: error conditions, such as the failure to replicate parts after an acceptable period; and alert conditions, such as the cluster becoming close to its storage limit or computers becoming unresponsive. In the month or so that the watchdogs have so far been deployed they have reported two alerts and no errors. When we have more confidence in the watchdogs we may integrate them into an automatic failure mitigation system to reduce the cluster management overhead.

```
void AddStorageComputer(string
    storageComputerName, ulong freeSpace,
    string managedDirectory, string
    uncPath, string httpPath, int
    networkId);

bool DeleteStorageComputer(string
    storageComputerName);

StorageComputerInfo
    GetStorageComputerInformation(string
    storageComputerName);

void SetFreeSpace(string
    storageComputerName, ulong freeSpace);

void SetStorageComputerState(string
    storageComputerName,
    StorageComputerState
    storageComputerState);

string[] ListStorageComputers();

ulong[] ListPartsAtStorageComputer(string
     storageComputerName);

ulong[] GetPartReplicaList(string
    storageComputerName);
```

Figure 3: TidyFS API - Operations involving storage computers

3.6 API

For completeness we list the TidyFS API here. Most operations involve storage computers, streams and parts. As previously described, these operations are used by other TidyFS components (node service and TidyFS Explorer) and by external applications that wish to read and write TidyFS data.

Figure 3 includes a representative set of operations involving storage computers. These include commands, typically used by cluster administrators, for adding, modifying and removing computers. `SetFreeSpace` and `SetStorageComputerState` are useful for updating the state of the cluster, and are used both by admin-

```
string[] ListDirectories(string path);

string[] ListStreams(string path);

void CreateStream(string streamName);

ulong CreateStream(string streamName,
    DateTime leaseTime, int numParts);

void CopyStream(string srcStreamName,
    string destStreamName);

void RenameStream(string srcStreamName,
    string destStreamName);

bool DeleteStream(string streamName);

void ConcatenateStreams(string
    destStreamName, string srcStreamName);

void AddPartToStream(ulong[] partIds,
    string streamName, int position);

void RemovePartFromStream(ulong partId,
    string streamName);

ulong[] ListPartsInStream(string
    streamName);

PartInfo[] GetPartInfoList(string
    streamName);

DateTime GetLease(string streamName);

void SetLease(string streamName, DateTime
     lease);

ulong RequestPartIds(string streamName,
    uint numIds);

byte[] GetStreamBlobAttribute(string
    streamName, string attrName);

void SetStreamAttribute(string streamName,
     string attrName, byte[] attrValue);

void RemoveStreamAttribute(string
    streamName, string attrName);

string[] ListStreamAttributes(string
    streamName);
```

Figure 4: TidyFS API - Operations involving streams

```
void AddPartInfo(PartInfo[] pis);

void AddPartReplica(ulong partId, string
    nodeName);

void RemovePartReplica(ulong partId,
    string nodeName);

void GetReadPaths(ulong partId, string
    nodeName, out StringCollection paths);

void GetWritePaths(ulong partId, string
    nodeName, out StringCollection paths);
```

Figure 5: TidyFS API- Operations involving parts

istrators and by the node service. Operations that allow listing all computers and all parts at a computer are also provided for diagnostic purposes and are used by the TidyFS Explorer. `GetPartReplicaList` is a command for listing all new replicas that have been assigned by TidyFS to a specific storage computer, but that have not yet been created. This call is invoked periodically by the node service running on a storage computer to fetch the list of parts that it needs to fetch and replicate.

Figure 4 lists operations involving streams. Due to space restrictions we have only included the most often used ones and have omitted some operations that are similar to others already shown. For example, while we only show operations for getting and setting stream attributes of blob data type, similar commands exist for attributes of different types. The figure includes operations for listing the contents (both subdirectories and streams) of directories, and stream manipulation commands including operations for adding and removing parts from a stream. Parts may be added at any position in the stream, and streams may be concatenated, which causes all parts from one stream to be appended to another.

Finally, in Figure 5 the remaining operations involving parts are listed. The `AddPartInfo` command is used by clients to inform TidyFS that a part has been fully written, and to pass information such as the part size and fingerprint. Operations for adding and removing replicas from a storage computer (`AddPartReplica` and `RemovePartReplica`) are used by the node service to inform the metadata server when replicas have been created at a particular computer. Other important operations include `GetReadPaths` and `GetWritePaths`, which return a list of paths where a part may be read from, or written to. These are used by clients prior to reading or writing any data to TidyFS. There are also operations for

getting and setting part attributes of various data types that are similar to those provided for streams and are omitted here for brevity.

The RSL state replication library allows some API calls to execute as "fast reads" which can run on any replica using its current state snapshot and do not require a round of the consensus algorithm. Fast reads can in principle reflect quite stale information, and are not serializable with other state machine commands. Given our read-mostly workload, by far the most common API calls are those fetching information about existing streams, such as listing parts in a stream or read paths for a part. We therefore allow these to be performed as fast reads and use all replicas to service these reads, reducing the bottleneck load on the metadata server. As we report in Section 4 most reads are from streams that have existed for a substantial period, so staleness is not a problem. If a fast read reports stale data such as a part location that is out of date following a replication, the client simply retries using the slow-path version that guarantees an up to date response.

4 Evaluation and experience

TidyFS has been deployed and actively used for the past year on a research cluster with 256 servers, where dozens of users run large-scale data-intensive computations. The cluster is used exclusively for programs run using the DryadLINQ [28] system. DryadLINQ is a parallelizing compiler for .NET programs that executes programs using Dryad [16]. Dryad is a coarse-grain dataflow execution engine that represents computations as directed-acyclic graphs of processes communicating via data channels. It uses TidyFS for storage of input and output data, following the access patterns set out in Section 2.2. Fault-tolerance is provided by re-execution of failed or slow processes. Dryad processes are scheduled by a cluster-wide scheduler called Quincy [17] that takes into account the size and location of inputs for each process when choosing which computer to run the process on. The cluster hardware is as described in our Quincy paper [17].

Dryad queries TidyFS for the locations of its input parts and passes this information to Quincy so that most processes end up scheduled close in network topology to at least one replica of their input parts. DryadLINQ makes use of TidyFS attributes to store type information about the records in output streams and to record partitioning information—both of these types of attribute are used by subsequent DryadLINQ programs that consume the streams as inputs. Our cluster infrastructure also include Nectar [12] which is a cache manager for computations. Nectar and DryadLINQ communicate to determine sub-computations whose results have already been stored to TidyFS, to save unnecessary recomputation of complex quantities. Nectar makes use of TidyFS stream and part fingerprints to determine when the inputs of two computations are identical.

TidyFS was designed in conjunction with the other cluster components listed above, so it naturally has APIs and performance that is well suited to their requirements. As noted in Section 2.2 we believe these requirements are shared by other systems including MapReduce and Hadoop. At the end of the paper we include more discussion of the wider applicability of TidyFS.

4.1 Data volume

On a typical day, several terabytes of data are read and written to TidyFS through the execution of DryadLINQ programs. We collected overall statistics on the usage of the system through logs maintained by TidyFS.

Figures 6 and 7 present daily read and write loads, and volumes of data deleted, on TidyFS during a sample two-week period. The purpose of these figures is to give the reader a sense of the scale of system usage. The amount of data read and written on a specific day varies over the days mostly because of the diverse nature of jobs being run on the cluster, and on other factors such as day of the week.

As earlier described, Quincy attempts to place processes close to the data they will read, and when satisfying clients' requests for read paths TidyFS prioritizes copies of the data that are stored locally at the computer where the client is running, followed by copies stored within the same rack, and finally cross racks. The default replication factor in the cluster is two, so there are generally two replicas per part from which data can be read. Figure 6 classifies reads as local, within rack, cross rack or remote. Remote reads refer to reads where the client is outside the compute cluster. As expected, the majority of reads are local, followed by reads that occur within the same rack, indicating that the goal of moving computation close to the data is very often achieved.

DryadLINQ does not perform any eager replication, so each part is written once by DryadLINQ and subsequently replicated lazily by TidyFS. Figure 7 shows the amount of data committed by DryadLINQ per day during the sample period. The vast majority of these writes are local, and there is an equivalent amount of data written during lazy replication since the default replication count is two. The volume of data deleted shown in the figure corresponds again to the volume of primary part data: due to the replication factor of two, the space freed across the cluster disks is actually twice as large.

4.2 Access patterns

More insight can be gained into cluster usage by studying how soon data is read after it has been initially written.

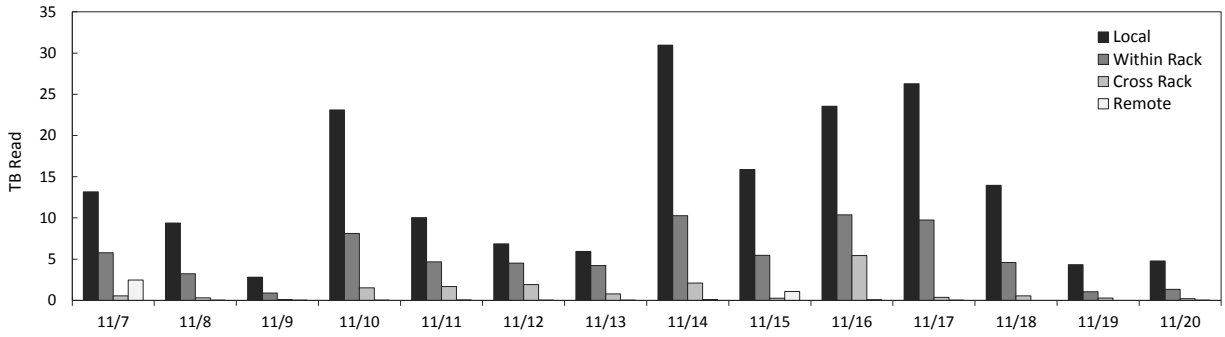


Figure 6: Terabytes of data read per day, grouped by local, within rack, cross-rack, and remote reads.

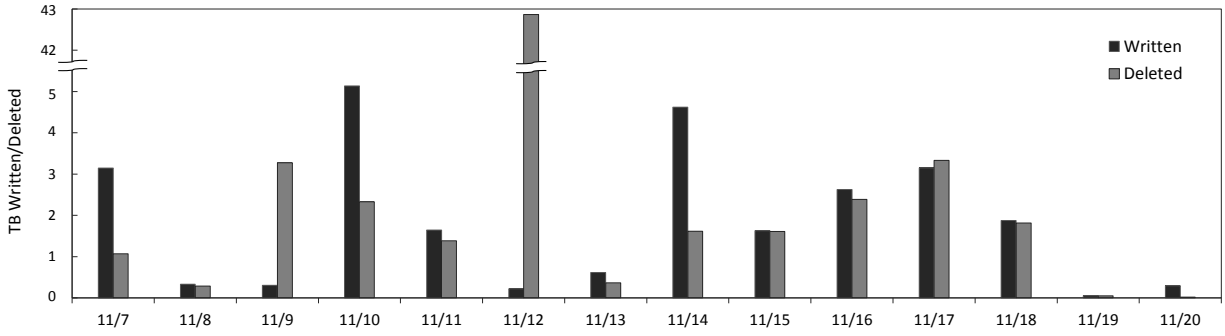


Figure 7: Terabytes of primary part data written and deleted per day.

We computed, for every read that happened over a period of three months, the age of the read as the time interval between the time the read occurred and the time the part being read was originally written. Figure 8 presents the cumulative distribution of data read as the read age increases. As shown in the figure only a small percentage of data is read within the first five minutes after it has been created. Almost 40% of the data is read within the first day after the data being read has been created, and around 80% of the data is read within one month of creation.

Given the small percentage of reads that occur within the first minute of writing a part, the node service's periodic task of checking for pending replicas is configured to run every minute. This implies a delay of up to a minute before lazy replication of a part begins, and reads that occur in smaller windows of time will have fewer choices of where to read from.

This effect can be observed in Figure 9, which shows the proportion of local, within rack and cross rack data out of all reads that happen for different read ages. As observed, most reads that happen within the first minute after a part is written are remote reads, since many of those parts would only have one copy at that time. However, as observed from Figure 8 the total number of reads that happen at that time interval is very small relative to all reads. For part reads that occur after longer periods of time since the part's creation, local and within rack reads predominate.

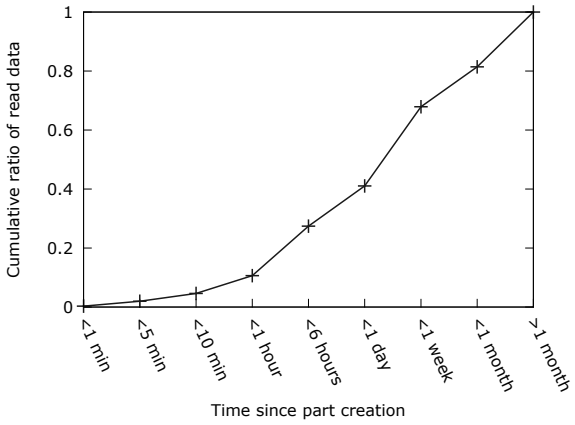


Figure 8: Cumulative distribution of read ages (time when read occurs - time when data was originally written) for reads occurring over a period of three months.

To further characterize our cluster's usage pattern we analyzed the relationships in timing and frequency between reads and writes of the same part. In Figure 10 we show how often parts are read once they have been written. Once again, we analyzed data over a period of three

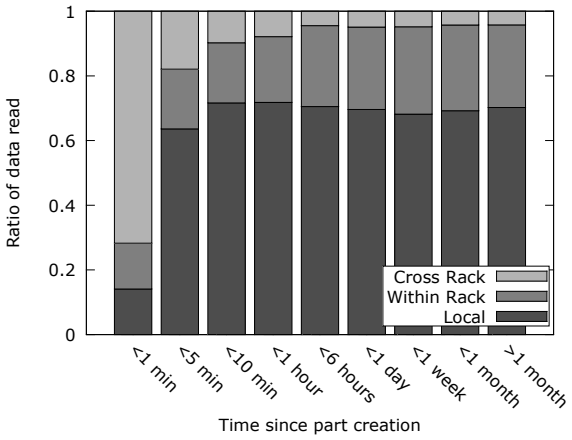


Figure 9: Proportion of local, within rack and cross rack data read grouped by age of reads.

months, considering all writes and subsequent reads that happened in the period. For each part written we counted the number of times the part was subsequently read. As observed from the graph, many parts are read only once or a small number of times. There is also a large number of parts which are never read again by future jobs.

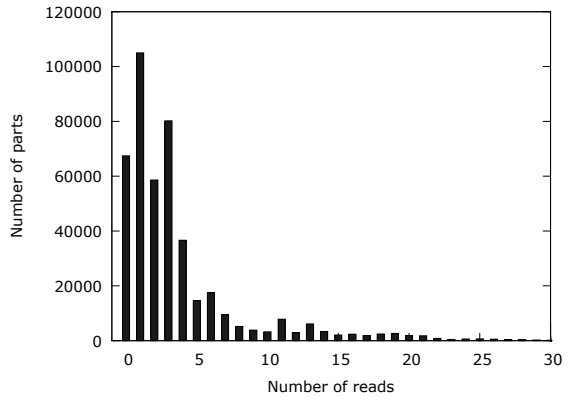


Figure 10: Number of times a part is read after it has been originally written.

Finally, in Figure 11 we present results on the last access time of parts. For every part in the system, we identify the last time it was read, up to a maximum of sixty days, and plot the cumulative ratio of parts as a function of their last read time. Approximately thirty percent of the parts had not been read in the period of sixty days, and read ages are evenly distributed over the sixty day period.

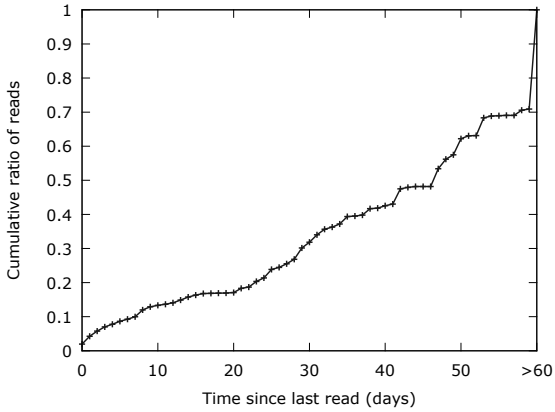


Figure 11: Cumulative distribution of parts over time since parts' last access.

4.3 Lazy versus Eager Replication

In order to evaluate the effectiveness of lazy replication, we gathered statistics about the average time before parts are replicated. Table 1 shows the mean time between a part being added to the system and the creation of a replica for that part over a three month time window. Nearly 70% of parts have a replica created within 60 seconds and 84% within 2 minutes. 96% of parts have a replica within one hour. Parts that are not replicated within one hour are due to the node service on the storage computer where the replica has been scheduled being disabled for system maintenance. The data in these parts is still available from the original storage computer. Therefore, we find that lazy replication provides acceptable performance for clusters of a few hundred computers. We have been experiencing around one unrecoverable computer failure per month, and have not so far lost any unreplicated data as a consequence.

Mean time to replication (s)	Percent
0 - 30	6.7%
30 - 60	62.9%
60 - 120	14.6%
120 - 300	1.1%
300 - 600	2.2%
600 - 1200	4.5%
1200 - 3600	3.4%
3600 -	4.5%

Table 1: Mean time to replication over a three month time interval.

4.4 Replica Placement and Load Balancing

As described in Section 3.4, we would like TidyFS to assign replicas to storage computers using a policy that balances the spread of parts per stream across computers as well as the total free space available at storage computers. We compare the two policies we implemented: the initial space-based policy that led to poorly-balanced streams, especially for those streams with many small parts; and the subsequent best-of-three random choice policy.

We define a load-balancing coefficient for each stream by calculating the L^2 distance between a vector representing the number of parts from a particular stream assigned to a specific storage computer and the perfectly-balanced mean vector. The coefficient is computed as follows: $\sqrt{\sum_{i=1}^{n}(p_i - \frac{rp}{n})^2}$ where r is the stream replication factor, p is the number of parts in the stream, p_i is the number of part replicas stored at node i, and n is the number of computers in the `ReadWrite` state in the cluster. We normalize so that a coefficient of 1 corresponds to the worst-case situation where just r computers are used to store all the parts, which leads to the following complete equation:

$$\frac{1}{\sqrt{r(p-\frac{rp}{n})^2 + (n-r)(\frac{rp}{n})^2}} \sqrt{\sum_{i=1}^{n}(p_i - \frac{rp}{n})^2} \quad (1)$$

The load-balancing behavior was analyzed over two periods of time: in the first one, the space-based policy was used; in the second one, the randomized policy. We computed, at the end of each day, the load balancing coefficient of each stream, as given by Equation 1, and the overall average over all streams. Figure 12 presents the average coefficient for each day over these two periods. As shown in the figure, streams were significantly better balanced during the second period when the randomized policy was being used.

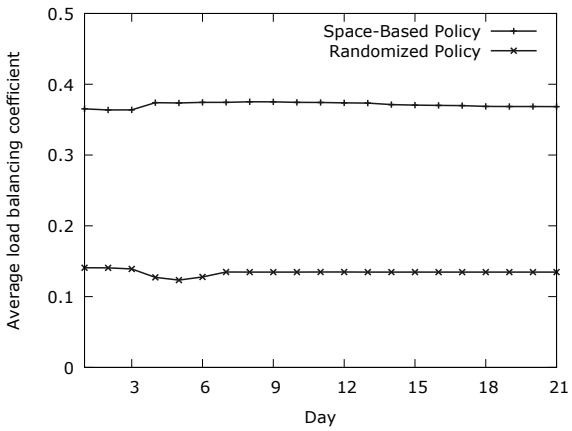


Figure 12: Load balancing coefficient when using space-based policy against randomized policy for replica placement.

5 Related Work

The TidyFS design shares many characteristics with other distributed storage systems targeted for data-intensive parallel computations, but wherever possible simplifies or removes features to improve the overall performance of the system without limiting its functionality for its target workload. While several systems only focus on delivering aggregate performance to a large number of clients, one of the main goals with TidyFS is to also encourage moving computation close to the data by providing interfaces that allow applications to locate all replicas of parts.

Like several distributed storage systems such as Frangipani [26], GPFS [23], GFS [11], HDFS [6, 24], PanasasFS [27] and Ursa Minor [25], TidyFS separates metadata management from data storage, allowing data to be transparently moved or replicated without client knowledge. Some of these systems do not maintain the metadata in centralized servers and support complex read and write semantics, either by relying on distributed locking algorithms (Frangipani, GPFS), or by migrating metadata prior to executing metadata operations (Ursa Minor).

TidyFS is most similar to GFS [11] and HDFS [6, 24]. It follows their centralized metadata server design and focuses on workloads where data is written once and read multiple times, which simplifies the needed coherency model. Despite the similarities with GFS and HDFS, TidyFS's design differs from these systems in important ways. The most significant difference is that TidyFS uses native interfaces to read and write data while GFS and HDFS both supply their own data access APIs. This design choice leads to related differences in, for example, replication strategies and part-size distribution.

TidyFS also differs from GFS in how it achieves resilience against metadata server crashes. GFS maintains checkpoints and an operation log to which metadata operations should be persisted before changes become visible to clients. TidyFS instead maintains multiple metadata servers, all of which keep track of all the metadata in the system, and uses the Paxos [18] algorithm to maintain consistency across the servers. A similar approach is used in BoomFS [2], a system similar to HDFS built from declarative language specifications.

TidyFS's ability to handle database parts enables a hybrid large-scale data analysis approach that exploits the performance benefits of database systems, similarly to the approach taken for HadoopDB [1]. HadoopDB combines MapReduce style computations with database systems to achieve the benefits of both approaches, although

the databases accessed by HadoopDB are not stored in HDFS or a replicated file system. The MapReduce framework is used to parallelize queries, which are then executed on multiple single-node database systems. Database parts stored in TidyFS can be queried using Dryad and DryadLINQ in similar ways.

6 Discussion

TidyFS is designed to support workloads very much like those generated by MapReduce and Hadoop. It is thus natural to compare TidyFS to GFS and HDFS, the file systems most commonly used by MapReduce and Hadoop respectively. The most consequential difference is the decision for TidyFS to given clients direct access to part data using native interfaces. Our experience of the resulting simplicity and performance, as well as the ease of supporting multiple part types such as SQL database, has validated our assumption that this was a sensible design choice for the target Dryad workload. The main drawback is a loss of generality. Other systems built on GFS, such as BigTable [8], use the ability to persist small appends and make them visible to other clients in order to achieve performance and reliability, and the desire to support appends in HDFS is related to the desire to implement similar services such as HBASE on top of that file system [10]. A key point in [20] is that the GFS semantics were not a good fit for all of the applications rapidly built on GFS. Some issues that are described in [20], such as the small file problem, can be addressed in client libraries. Other issues, including inconsistent data returned to the client depending on which replica was read and latency issues because GFS was designed for high-throughput, not low-latency, cannot be addressed in client libraries. We believe however that rather than complicating the common use case of a data-intensive parallel file system it makes more sense to add a separate service for reliable logging or distributed queues. This was done for example in River [4] and the Amazon Web Service [3] and would be our choice if we needed to add such functionality to our cluster.

Another feature that TidyFS lacks as a result of our choice of native interfaces is automatic eager replication, with the exception of optional eager replication in the data ingress case. Again we are happy with this tradeoff. In the year we have been operating TidyFS we have not had a single part lost before replication has completed. Clearly this is primarily because of the relatively small size of our deployment, however it suggests that leveraging the client's existing fault-tolerance to replace lost data is a reasonable alternative to eager replication, despite the additional work spent in the rare failure cases.

The final major difference is our lack of control over part sizes. DryadLINQ programs frequently make use of the ability to output streams with exact, known partitioning, which leads to sometimes significant performance improvements. However we do also have to deal with problems caused by occasional parts which are very much larger than the average. This caused problems with our original simple replication policy that fortunately were easy to fix with the slightly more sophisticated best-of-three random policy. We believe that the existence of very large parts also adds to disk fragmentation across our cluster. If ignored, we have found that this fragmentation results in devastating performance penalties as parts are split into thousands of fragments or more, preventing the sequential I/O that is necessary for high read throughput. We have recently started to aggressively defragment all disks in the cluster to mitigate this problem.

While we motivate the TidyFS design using general properties of data intensive shared-nothing workloads, in practice it is currently used almost exclusively by applications executing using Dryad. Rather than making TidyFS more general, one direction we are considering is integrating it more tightly with our other cluster services. If all I/O on the cluster were reads or writes from TidyFS, Dryad intermediate data shuffling, and TidyFS replication traffic, then substantial performance benefits might accrue from integrating I/O into the Quincy scheduling framework, and possibly even adopting circuit-switched networking hardware to take advantage of these known flows. As mentioned in Section 2.2 this tighter integration might conflict with the choice to allow clients unfettered access to native I/O. On the other hand if the only client were Dryad, which is trusted to obey scheduling decisions, the benefits of I/O scheduling might still be achieved. Tighter integration with Dryad would also let us revisit a design alternative we had originally considered, which is to eliminate the node service altogether and perform all housekeeping tasks using Dryad programs. We abandoned this potentially simplifying approach primarily because of the difficulty of ensuring that Dryad would run housekeeping tasks on specific computers in a timely fashion with its current fairness and locality policies.

We have recently reimplemented the metadata server on top of a replicated SQL database instead of the C++ and RSL implementation described in this paper. This radically reduces the number of lines of novel code in the system by relying on the extensive but mature SQL Server codebase. Our main concern is whether comparable performance can be easily attained using SQL, which is unable to perform fast reads as described in Section 3.6 without the addition of a custom caching layer, and we are currently evaluating this tradeoff.

Overall we are pleased with the performance, simplicity and maintainability of TidyFS. By concentrating on a single workload that generates a very large amount of I/O traffic we were able to revisit design decisions made

by a succession of previous file systems. The resulting simplifications have made the code easier to write, debug, and maintain.

Acknowledgments

We would like to thank Mihai Budiu, Jon Currey, Rebecca Isaacs, Frank McSherry, Marc Najork, Chandu Thekkath, and Yuan Yu for their invaluable feedback, both on this paper and for many helpful discussions about the TidyFS system design. We would also like to thank the anonymous reviewers and our shepherd, Wilson Hsieh, for their helpful comments.

References

[1] ABOUZIED, A., BAJDA-PAWLIKOWSKI, K., ABADI, D. J., SILBERSCHATZ, A., AND RASIN, A. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In *Proceedings of the 35th Conference on Very Large Data Bases (VLDB)* (Lyon, France, 2009).

[2] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (Paris, France, 2010).

[3] Amazon Web Services. http://aws.amazon.com/.

[4] ARPACI-DUSSEAU, R. H., ANDERSON, E., TREUHAFT, N., CULLER, D. E., HELLERSTEIN, J. M., PATTERSON, D., AND YELICK, K. Cluster I/O with River: making the fast case common. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems (IOPADS)* (Atlanta, GA, USA, 1999).

[5] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proceedings of the Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (San Diego, CA, USA, 2007).

[6] BORTHAKUR, D. HDFS architecture. Tech. rep., Apache Software Foundation, 2008.

[7] BRODER, A. Some applications of rabin's fingerprinting method. In *Sequences II: Methods in Communications, Security, and Computer Science* (1993), Springer Verlag, pp. 143–152.

[8] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI))* (Seattle, WA, USA, 2006).

[9] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, USA, 2004).

[10] File appends in HDFS. http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs/.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP)* (Bolton Landing, NY, USA, 2003).

[12] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in data centers. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, 2010).

[13] Hadoop wiki. http://wiki.apache.org/hadoop/, April 2008.

[14] The HIVE project. http://hadoop.apache.org/hive/.

[15] ISARD, M. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 60–67.

[16] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of 2nd European Conference on Computer Systems (EuroSys)* (Lisbon, Portugal, 2007).

[17] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, USA, 2009).

[18] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[19] MACCORMICK, J., MURPHY, N., RAMASUBRAMANIAN, V., WIEDER, U., YANG, J., AND ZHOU, L. Kinesis: A new approach to replica placement in distributed storage systems. *ACM Trans. Storage 4*, 4 (2009), 1–28.

[20] MCKUSICK, M. K., AND QUINLAN, S. GFS: Evolution on fast-forward. *Queue 7* (August 2009), 10:10–10:20.

[21] MITZENMACHER, M., RICHA, A. W., AND SITARAMAN, R. The power of two random choices: A survey of techniques and results. In *Handbook of Randomized Computing* (2001), Kluwer, pp. 255–312.

[22] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 28th International Conference on Management of Data (SIGMOD)* (Vancouver, Canada, 2008).

[23] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST)* (Monterey, CA, 2002).

[24] SHVACHKO, K. V. HDFS scalability: The limits to growth. *;login 35*, 2 (April 2010).

[25] SINNAMOHIDEEN, S., SAMBASIVAN, R. R., HENDRICKS, J., LIU, L., AND GANGER, G. R. A transparently-scalable metadata service for the Ursa Minor storage system. In *Proceedings of the USENIX Annual Technical Conference (USENIX)* (Berkeley, CA, USA, 2010).

[26] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: a scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP)* (Saint Malo, France, 1997).

[27] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable performance of the Panasas parallel file system. In *Proceedings of the 6th Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, 2008).

[28] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, USA, 2008).

Eyo: Device-Transparent Personal Storage

Jacob Strauss[*] *Justin Mazzola Paluska*
Chris Lesniewski-Laas *Bryan Ford*[†] *Robert Morris* *Frans Kaashoek*

Massachusetts Institute of Technology [†]*Yale University*
[*]*Quanta Research Cambridge*

Abstract

Users increasingly store data collections such as digital photographs on multiple personal devices, each of which typically offers a storage management interface oblivious to the contents of the user's other devices. As a result, collections become disorganized and drift out of sync.

This paper presents *Eyo*, a novel personal storage system that provides *device transparency*: a user can think in terms of "file X", rather than "file X on device Y", and will see the same set of files on all personal devices. *Eyo* allows a user to view and manage the entire collection of objects from any of their devices, even from disconnected devices and devices with too little storage to hold all the object content. *Eyo* synchronizes these collections across any network topology, including direct peer-to-peer links. *Eyo* provides applications with a storage API with first-class access to object version history in order to resolve update conflicts automatically.

Experiments with several applications using *Eyo*—media players, a photo editor, a podcast manager, and an email interface—show that device transparency requires only minor application changes, and matches the storage and bandwidth capabilities of typical portable devices.

1 Introduction

Users often own many devices that combine storage, networking, and applications managing different types of data: e.g., photographs, music files, videos, calendar entries, and email messages. When a single user owns more than one such device, that user needs a mechanism to access their objects from whichever device they are using, in addition to the device where they first created or added the object to their collection. Currently, users must manually decide to shuttle all objects to a single master device that holds the canonical copy of a user's object collection. This hub-and-spoke organization leads to a storage abstraction that looks like "object a on device x", "object b on device y", etc. It is up to the user to keep track of where an object lives and determine whether a and b are different objects, copies of the same object, or different versions of the same object.

A better approach to storing personal data would provide *device transparency*: the principle that users should see the same view of their data regardless of which of their devices they use. Device transparency allows users to think about their unified data collection in its entirety regardless of which device a particular object may reside on, rather than as the union of independent copies of objects scattered across their devices.

Traditional distributed file systems provide *location transparency* whereby a file's name is independent of its network location. This property alone is insufficient for use with disconnected, storage-limited devices. A device-transparent storage system, however, would provide the same abstraction regardless of connectivity.

One attempt at providing device transparency is to store all data on a centralized cloud server, and request objects on demand over the network. In the presence of poor or disconnected networks, however, this approach fails to provide device-transparency: disconnected devices cannot access new objects or old objects not cached locally. In addition, two devices on the same fast local network cannot directly exchange updates without communicating with the central hub.

Beyond the challenge of transferring data between devices, either by direct network connections or via centralized servers, providing device-transparent access to a data collection faces two additional challenges: (1) concurrent updates from disconnected devices result in conflicting changes to objects, and (2) mobile devices may not have enough space to store an entire data collection.

This paper presents *Eyo*, a new personal storage system that provides device transparency in the face of disconnected operation. *Eyo* synchronizes updates between devices over any network topology. Updates made on one device propagate to other reachable devices, and users see a single coherent view of their data collection from any of their devices. Since these updates may cause

conflicts, *Eyo* supports *automated conflict resolution*.

The key design decision behind *Eyo* is to use object metadata (e.g., author, title, classification tags, play count, etc.) as a proxy for objects themselves. This decision creates two requirements. First, *Eyo* requires applications to *separate object metadata from content*, so that *Eyo* knows what is metadata and what is content. Second, *Eyo* must replicate metadata on *every* device, so that applications can manage any object from any device as though that device held the master copy of that object.

To meet these requirements, *Eyo* provides a new storage API to applications. This API separates metadata from content, and *presents object version histories as first-class entities*, so that applications can automatically resolve most common divergent version histories without user intervention, while incorporating the presentation and resolution of other conflicts as a part of ordinary operation. In return, applications delegate inter-device synchronization to *Eyo*.

Experiments using *Eyo* in existing applications—media players, a photo editor, a podcast manager, and an email interface—show that *Eyo* transforms these standalone applications into distributed systems providing device-transparent access to their data collections, takes advantage of local peer-to-peer communication channels, permits automatic conflict resolution, and imposes only modest storage and bandwidth costs on devices.

Eyo's main contribution is a design for device transparency for disconnected storage-limited devices, building on our earlier proposal for device transparency [44]. The design adopts techniques pioneered by existing systems (e.g., disconnected operation in Coda [22], application-aware conflict resolution in Bayou [46], placement rules in Cimbiosys [36] and Perspective [40], version histories in source control systems [16], update notifications in EnsemBlue [33]). Eyo wraps these techniques in a new storage interface that supports efficient, continuous, peer-to-peer synchronization, and avoids most user involvement in conflict resolution.

The remainder of this paper is organized as follows: Section 2 describes *Eyo*'s API and its use, followed by *Eyo*'s synchronization protocols in Section 3 and implementation in Section 4. Section 5 evaluates *Eyo* with existing data collections and applications. Section 6 describes related systems, and Section 7 concludes.

2 *Eyo*

Eyo enables a traditionally architected, single-device application to work as a distributed application whose state is scattered among many devices. For example, suppose we have a traditional photo management application that copies photos from a camera into a local database of photo albums. After modifying the application to use *Eyo*, the album database becomes replicated automatically across all devices, permitting the user to manage her photo collection from whichever device is most convenient. *Eyo* maintains these properties for the photo application on all devices, even if a given device isn't large enough to hold the entire collection of photos.

Eyo sits between applications, local storage, and remote network devices. *Eyo* uses an overlay network [13] to identify a user's devices, and track them as they move to different locations and networks. *Eyo* manages all communication with these devices directly, and determines which updates it must send to each peer device whenever those devices are reachable.

In order to bring these features to applications, *Eyo* provides a new storage API. *Eyo*'s API design makes the following assumptions about applications:

- Users identify objects by metadata, not filesystem path. For example, headers and read/replied flags of emails or the labels and dates of photos.
- The application provides user interfaces that make sense when the device stores only object metadata; for example, songs listings or genre searches.
- Modification of metadata and insertion/deletion of objects are common, but modification of object content is rare. For example, a user is more likely to change which folder a stored email message resides in, and less likely to change the message itself.
- Metadata is small enough to replicate on every device. In our application study (Section 5.1), we find that metadata is less than 0.04% of the size of typical music and photo objects.
- Application developers agree on the semantics of a basic set of metadata for common data types, in order to permit multiple applications to share the same data objects: e.g., standard email headers, ID3 tags in MP3 audio files, or EXIF tags in photos. Applications can still attach arbitrary data to metadata in addition to the commonly agreed upon portions.

We believe that these assumptions match the characteristics of common personal data management applications. The following sections describe how *Eyo*'s techniques can transform such applications into peer-to-peer distributed applications operating on a device-transparent data collection, using the photo album application as a running example.

2.1 Objects, metadata, and content

Eyo stores a user's object collection as a flat set of versioned objects. Figure 1 shows an example of an object representing one photo, with multiple versions from adding and removing organizing tags. Each *Eyo* version consists of a directed acyclic graph of collections of metadata and content. Edges in the version graph denote parent-child relationships. Newly created object versions include explicit predecessor pointers to their parent ver-

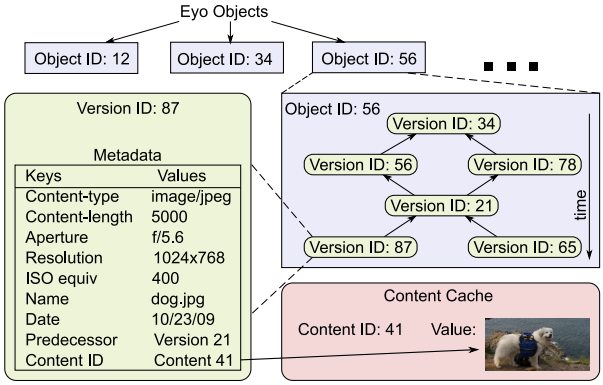


Figure 1: *Eyo* object store.

object creation and manipulation:

create(*ID hint*) → (*objectID*, *versionID*)
lookup(*query*) → list<(*objectID*, *versionID*)>
getVersions(*objectID*) → list<*versionID*>
getMetadata(*objectID*, *versionID*) → list<(*key*,*value*)>
open(*objectID*, *versionID*) → *contentID*
read(*contentID*, *offset*, *length*) → *contents*
newVersion(*objectID*, list<*versionID*>,
metadata, *contents*) →*versionID*
deleteObject(*objectID*) → *versionID*

placement rules:

addRule(*name*, *query*, *devices*, *priority*) → *ruleID*
getRule(*name*) → (*ruleID, query, devices, priority*)
getAllRules() → list<(*ruleID, query, devices, priority*)>
removeRule(*ruleID*)

event notifications:

addWatch(*query*, *watchFlags*, *callback*) → *watchID*
removeWatch(*watchID*)
callback(*watchID*, *event*)

Figure 2: *Eyo* API summary. Event notifications are discussed in Section 2.2, and placement rules in Section 2.3.

sions, represented as a *parent* version attribute. An object version's metadata consists of a set of *Eyo*- and application-defined key/value pairs. The metadata also contains a content identifier; the associated content might or might not be present on any particular device.

Applications retrieve objects via queries on metadata. *Eyo* expects applications to maintain rich enough metadata to display to the user meaningful information about an object, even on devices not storing the content. In our photo album example, the metadata may include rating, album, and location to help the user sort photos.

Eyo's API contains elements similar to databases for searching, reading, and editing object metadata, along with elements similar to filesystems for reading and writing object content. This distinction is deliberate, as it matches common uses of media applications which often use both elements internally. In addition, the API provides mechanisms to learn about and repair conflicts, to specify content placement rules, and to receive notices about changes to the object store. Figure 2 lists commonly used *Eyo* methods. The figure omits alternate iterator-based versions of these methods for constructing or viewing large collections as well as library functions combining these base operations. All of these methods access only device-local data, so no method calls will block on communication with remote devices.

If an application tries to read an object's content, but the content is not present on the device, *Eyo* signals an error. A user can still perform useful operations on metadata, such as classifying and reorganizing objects (e.g., updating the rating of a photo), from a device that does not store content. If the user wants to use content that is not on the current device, the system can use the metadata to help the user find a device that has the content, or ask *Eyo* to try to fetch the content using the placement methods in the API (Section 2.3). Section 3 shows how metadata replication supports efficient synchronization.

2.2 Queries

While *Eyo* does not provide human-readable object identifiers, it provides queries with which applications can implement their own naming and grouping schemes. For example, the photo application may tag photos with their associated albums. Queries return IDs for all objects that have metadata attributes matching the query. As in Perspective [40], users never see *Eyo* queries; applications create queries on their behalf.

Eyo's lookup() call performs a one-time search, whereas addWatch() creates a persistent query. Watch queries allow applications to learn of new objects and object versions, and to observe the progress of inter-device synchronization, fulfilling a purpose similar to filesystem notification schemes such as inotify [28].

Eyo's queries use a subset of SQL, allowing boolean combinations of comparisons of metadata values with constants. Such queries are efficient to execute but limited in expressiveness. For example, the language does not directly support searching for the 10 most-viewed photos, but does allow searching for photos viewed more than 100 times. *Eyo* limits queries to these restricted forms to assure efficiency for query uses (watch events and placement rules) that must evaluate queries in two different contexts: evaluating new or changed queries to identify which objects match, and determining which existing queries match new or modified objects.

2.3 Placement Rules

Eyo allows applications to specify *placement rules* controlling which objects' content has highest priority for storage on storage-limited devices. For example, the placement rules for our photo album application may

specify that a user's laptop should hold only recent albums, but that a backup device should hold every photo. Applications are expected to generate placement rules based on user input. Experience suggests that users are not very good at predicting what objects they will need or at describing those objects with rules [40]. *Eyo*'s metadata-everywhere approach makes it easy to find missing objects by searching the metadata, to discover which devices currently have copies of the object, and to fix the placement rules for the future.

Applications specify placement rules to *Eyo* using the query language. A placement rule is the combination of a query and the set of devices that should hold objects matching the query. For example, the photo album application might present a UI allowing the user to indicate which devices should hold a complete photo album. An application can also let users specify particular objects and the devices on which they should be placed.

Each rule has a priority, and a storage-limited device stores high-priority content in preference to low-priority. When space permits, *Eyo* provides *eventual filter consistency* [36] for object content, meaning that each device eventually gathers the set of objects that best matches its preferences. *Eyo*'s synchronization mechanism, as described in Section 3.4, ensures that at least one copy of content persists even if no placement rule matches.

To ensure that all devices know all placement rules, *Eyo* stores each rule as an object with no content, but whose metadata contain the query, priority, and device set. Any device can modify a placement rule. If a conflict arises between rule versions, *Eyo* conservatively applies the union of all current versions' requirements. Similarly, if an object has multiple current versions and any current version matches a placement query, *Eyo* acts as if the query had matched *all* versions back to the common ancestor. This behavior ensures that any device that may be responsible for the object's content has all versions required to recognize and resolve conflicts.

Because placement rules operate at object granularity, applications that maintain related variations of content should store these variations as separate objects linked via metadata, so that different placement rules can apply to each variation. For example, our photo application stores both a full size and a thumbnail size image of the same base photo, assigning a high priority placement rule to replicate the thumbnail objects widely, but placing the full-size versions only on high-capacity devices.

2.4 Object Version Histories

Much of *Eyo*'s API design and storage model is motivated by potentially disconnected devices. Devices carry replicas of the *Eyo* object store and might make independent modifications to their local replicas. Devices must therefore be prepared to cope with divergent replicas.

When an *Eyo* application on a device modifies an object, it calls newVersion() to create a new version of that object's metadata (and perhaps content) in the device's data store. The application specifies one or more parent versions, with the implication that the new version replaces those parents. In the ordinary case there is just one parent version, and the versions form a linear history, with a unique latest version. *Eyo* stores each version's parents as part of the version.

Pairs of *Eyo* devices synchronize their object stores with each other, as detailed in Section 3. Synchronization replaces each device's set of object versions and metadata attributes with the union of the devices' sets.

For example, in Figure 1, suppose device A uses *Eyo* to store a new photo, and to do so it creates a new object $O56$, with one version, $O56{:}34$, and metadata and content for that version. If A and B synchronize, B's object store will then also contain the new object, its one version, that version's metadata, and perhaps its content. If an application on B then modifies $O56$'s metadata or content, the application calls newVersion(O56, [O56:34], *metadata*, *content*), indicating that the new version ($O56{:}78$), should supplant the existing version. When A and B next synchronize, A will learn about $O56{:}78$, and will know from its parent that it supersedes $O56{:}34$. Since the version history is linear, *Eyo* applications will use the unique most recent version.

2.5 Continuous Synchronization

To propagate updates to other devices as promptly as possible, *Eyo* provides *continuous synchronization*. Continuous synchronization helps reduce concurrency conflicts by propagating changes as quickly as the network allows, essentially serializing changes. Continuous synchronization also improves the user experience by showing changes from other devices "instantly". If two devices on the same network run the photo album application, for example, rating changes in one application will be immediately reflected in other application. Section 3 details continuous synchronization.

2.6 Automated Conflict Management

A primary goal of *Eyo*'s API is to enable applications to offer the user automated conflict management. To manage conflicts, applications need access to history information, notifications when conflicts arise, and a way to resolve those conflicts permanently.

Eyo uses per-object version histories and update notifications to provide a distributed metadata database that describes objects at the same granularity as user-visible objects. Applications thus need to examine only the *Eyo*-provided history of changes to a single object at a time in order to resolve changes. Cloud synchronization services that use existing filesystem APIs would instead require

applications to examine two (or more) complete copies of a metadata database and write a resolution procedure to operate on the entire collection at once.

Continuing with the example from Figure 1, consider a case where A had produced a new version of $O56$ before the second synchronization with B, such as adding additional `category` or `location` tags to the photo. In that case, both new versions would have parent version $O56{:}34$. After synchronization, A and B would both have two "latest" versions of $O56$ in their object stores. These are called *head* versions. When it detects concurrent updates, *Eyo* presents to the application each of the head versions along with their common ancestors.

Eyo's version graphs with explicit multiple parent versions are inspired by version control systems [16, 45]. Where version control systems keep history primarily for users to examine, *Eyo* instead uses version history to hide concurrency from users as much as possible. When combined with synchronization, version graphs automatically capture the fact that concurrent updates have occurred, and also indicate the most recent common ancestor. Many procedures for resolving conflicting updates require access to the most recent common ancestor. Since *Eyo* preserves and synchronizes complete version graphs back to those recent ancestors, applications and users can defer the merging of conflicting updates as long as they want. In order to ensure that parent pointers in object version histories always lead back to a common ancestor, *Eyo* transfers older versions of metadata before newer ones during synchronization [34].

Applications hold responsibility for handling concurrent updates of the same object on different devices, and should therefore structure the representation of objects in a way that makes concurrent updates either unlikely or easy to merge automatically. Applications must notice when concurrent updates arise, via *Eyo* queries or watch notifications. When they do occur, applications should either resolve conflicts transparently to the user, or provide ways for users to resolve them. This division allows *Eyo* to take advantage of fleeting network connectivity to transfer all new updates. Users avoid interruptions about irrelevant objects, and can wait until some more convenient time to merge conflicts, or perhaps ignore unimportant conflicts forever.

Eyo's version history approach permits many concurrent updates to be resolved automatically and straightforwardly by the application. For example, a user may move a photo between albums on one device, while changing the rating for the same photo on another device. Applications can arrange for these pairs of operations to be *composable*, e.g., ensuring that album tags and ratings can be set independently in the metadata. *Eyo* identifies these conflicting modifications, but the applications themselves merge the changes since applications know the uses of these attribute types, and so can determine the correct final state for these classes of concurrent changes.

Some concurrent updates, however, require user intervention to merge them into a single version. For example, a user might change the caption of a photo different ways on different devices. In such cases it is sufficient for *Eyo* to detect and preserve the changes for the user, either to fix at some later time or ignore entirely. Because *Eyo* keeps all of the relevant ancestor versions, it is simple for the application to show the user what changes correspond to each head version. All of *Eyo*'s API calls work regardless of whether an object contains an unresolved conflict, so it is up to applications as to whether they wish to operate on conflicted objects.

Applications may not intentionally create conflicts: when calling newVersion(), applications may list only head versions as predecessors. This requirement means that once a unique ancestor is known to all devices in a personal group, no version that came before the unique ancestor can ever be in conflict with any new written or newly learned version. *Eyo* can thus safely delete these older versions without affecting later conflict resolution. For example, in Figure 1, if all devices knew about version $O56{:}21$, that version is a *unique ancestor* for the object $O56$, and *Eyo* may *prune* the older versions $O56{:}34$, $O56{:}56$, and $O56{:}78$. Section 5.4 discusses storage costs when devices do not agree on a unique ancestor.

Applications permanently remove objects from *Eyo* via deleteObject(), which is just a special case of creating a new version of an object. When a device learns that a delete-version is a unique ancestor (or that all head versions are deletes, and seen by all other devices), *Eyo* deletes that object from the metadata collection.

3 Continuous Synchronization

Eyo needs to synchronize two classes of data between devices, metadata and content, and faces different needs for these classes. Metadata is usually small, and updates must be passed as quickly as possible in order to provide the appearance of device-transparency. The goal of *Eyo*'s metadata synchronization protocol is to produce identical metadata collections after synchronizing two devices.

Content, in contrast, can consist of large objects that change infrequently and take a long time to send over slow network links. Synchronizing content, unlike metadata, results in identical copies of individual objects, but not of the entire collections. The goal of content synchronization is to move objects to locations that best match placement policies.

Given the different needs for these two classes of data, *Eyo* uses different protocols for each class. Both run over UIA [13], an overlay network supporting direct peer-to-peer links as well as centralized cloud server topologies.

3.1 Metadata Synchronization

The primary goal of *Eyo*'s metadata synchronization protocol is to maintain identical copies of the entire metadata collection. This process must be efficient enough to run continuously: updates should flow immediately to devices connected to the same network. If connectivity changes frequently, devices must quickly identify which changes to send to bring both devices up to date.

The main approach that *Eyo* takes to synchronize metadata is to poll for changes whenever connectivity changes and to push notifications to reachable devices whenever a local application writes a new version of an object. *Eyo* identifies and organizes changes as they occur rather than iterating over the entire collection, allowing *Eyo* to quickly find the set of changed objects (among a much larger set of unchanged objects) at every synchronization opportunity.

Eyo groups multiple metadata updates into a permanent collection called a *generation*. Each generation is uniquely named by the device that created it and includes an id field indicating how many generations that device has created. A generation includes complete metadata updates, but only identifiers and new status bits for content updates. All synchronization occurs at the granularity of individual generations; each device that holds a copy of a given generation will have an identical copy.

A *generation vector* is a vector denoting which generations a device has already received. These vectors are equivalent to traditional version vectors [32], but named differently to avoid confusion with the versions of individual objects. For a personal group with n devices, each *Eyo* device updates a single n-element vector of $(device, id)$ tuples indicating the newest generation authored by $device$ that it holds.

Each device regularly sends getGenerations requests to other reachable devices. When local applications modify or create new objects (via newVersion calls), *Eyo* adds these uncommunicated changes to a *pending* structure, and attempts to contact reachable peers. With each of these requests, the client includes either its local generation vector, or the next generation vector it will write if it has new changes pending. When a device receives a reply, it incorporates the newly learned changes into its local data store, updates its generation vector accordingly, notifies applications about newly learned changes, and updates and applies placement rules to the newly learned changes.

When a device receives an incoming getGenerations request, it first gathers all pending changes, if any, into a new generation. It then identifies all the changes the other device lacks, and replies with those changes. If the request includes a generation vector with some component larger than the device handling the request knows about, the device queues a getGenerations request in the reverse direction to update itself, either immediately, or when next reachable if the request fails.

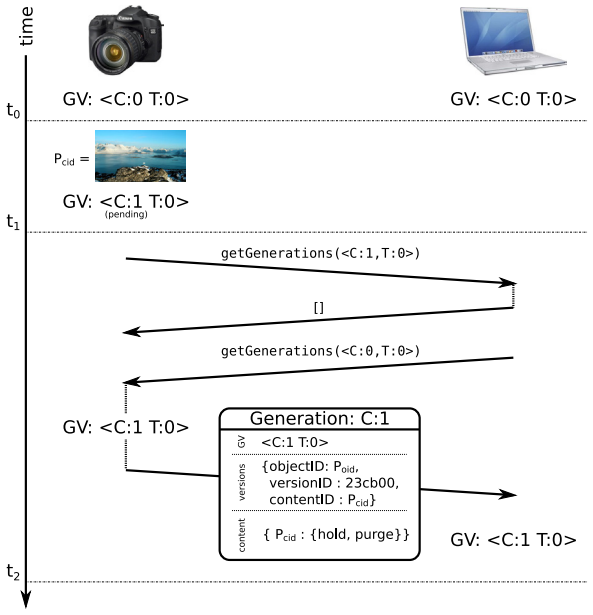


Figure 3: Metadata Synchronization: Messages sent between two devices for one new object

Figure 3 presents an example use of these structures between two devices: a camera C that temporarily stores photos when the user takes a picture, and a target device T that archives the user's photos. Initially, at t_0 in Figure 3, both devices hold no objects and agree on an initial generation vector `<C:0,T:0>`. When the user takes a picture P at time t_1, the camera adds the contents of the picture to its local content store with content identifier P_{cid}, creates a new *Eyo* object with object id P_{oid}, and adds P_{oid} to the metadata store. *Eyo* adds each of these updates to the next generation under construction (noted $pending$ in the figure).

At time t_2, C holds uncommunicated updates, so it sends getGenerations() requests to all reachable devices with the single argument `<C:1,T:0>`: C's generation vector with the `C` element incremented. T compares the incoming generation vector to its own and determines that it has no updates for C and replies with an empty generation list. However, since C's generation vector was larger than its own, T now knows that C has updates it has not seen, so T immediately makes its own getGenerations() call in the opposite direction with argument `<C:0,T:0>` since T has no uncommunicated updates of its own. Upon receiving the incoming request from T, C increments its generation vector and permanently binds all uncommunicated updates into generation `C:1`. C then replies with generation `C:1` and its newly-updated generation vector to T. The camera makes no

further call back to T, as T's generation vector was not larger than its own. Both devices now contain identical metadata.

Although for the sake of clarity this example only included two devices and did not include a large existing data collection, it does illustrate the protocol's scaling properties. For a group containing n devices, the *Eyo* metadata synchronization protocol sends only a single generation vector of length n to summarize the set of updates it knows about in a getGenerations() request. Upon receiving an incoming vector, an *Eyo* device needs only a simple lookup to identify what generations to send back, rather than an expensive search. This lookup requires one indexed read into the generation log per element in the incoming generation vector. This low cost means that devices can afford to push notifications instantaneously, and poll others whenever network connectivity changes.

3.2 History and Version Truncation

Eyo must have a way to prune version histories. It must identify which past changes are no longer needed and reclaim space taken up by those updates. This process involves three separate steps: determining when generation objects have been seen by all devices in a group, combining the contents of those generation objects into a single archive, and truncating the version history of individual objects.

Eyo learns that each other device has seen a given generation G by checking that every other device has written some other generation G' that includes G in its generation vector. At this point, no other existing device can correctly send a synchronization request that would include G in the reply, so it can remove G from its generation log. Once a device learns that all other devices have received a given generation G, it may lazily move G's contents into its *archive generation*, which groups together updates made by different devices and from different original generations, and does not retain those origins. *Eyo* preserves at least one generation for each device separate from the combined archive, even if that generation is fully known to all other devices. This ensures that *Eyo* knows the latest generation each other device has reported as received.

Object versions in the archive generation are known by all the user's devices, and are thus candidates for pruning, which is the second phase of history truncation. Version pruning then proceeds as described in Section 2.4.

3.3 Adding and Removing Devices

When a user adds a new device to their personal group, and that new device first synchronizes with an existing device, *Eyo* sees a getGenerations() request with missing elements in the incoming generation vector. Existing devices reply with a complete copy of all generations plus the archive generation. This copy cannot easily be broken down into smaller units, as the archive generation differs between devices due to pruning. Users expect new devices to require some setup, however, so this one-time step is not an undue burden.

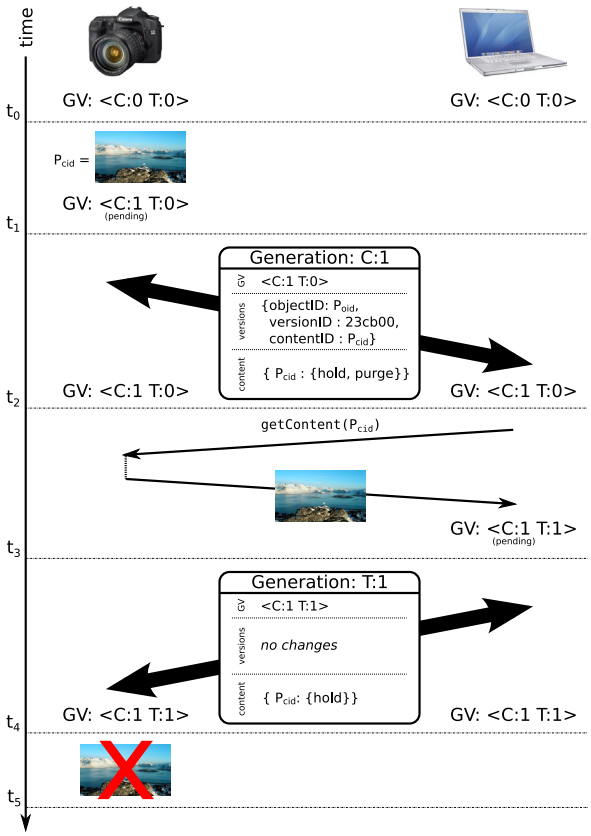


Figure 4: Content Synchronization. The thick double arrows represent a metadata sync from Figure 3.

Users remove devices from an *Eyo* group by deleting them from the underlying overlay network. Unless the user explicitly resets an expelled device entirely, it does not then delete any objects or content, and behaves thereafter as group with only one device. Removing an inactive or uncommunicative device from an *Eyo* group allows the surviving devices to make progress truncating history.

3.4 Content Synchronization

The challenges in moving content to its correct location on multiple devices are (1) determining which objects a particular device should hold, (2) locating a source for each missing data object on some other device, and (3) ensuring that no objects are lost in transit between devices.

Eyo uses placement rules to solve the first of these

challenges, as described in Section 2.3. Each device keeps a sorted list of content objects to fetch, and updates this list when it learns about new object versions, or when changes to placement rules affect the placement of many objects.

Eyo uses the global distribution of metadata through a user's personal group to track the locations of content objects. In addition to the version information, devices publish notifications about which content object they hold (as shown in Figure 3). Since all devices learn about all metadata updates, all devices thus learn which devices should hold content as part of the same process. When *Eyo* learns that another device is reachable, it can look at the list of content to fetch, and determine which objects to request from the reachable device.

To ensure that content objects are not deleted prematurely, *Eyo* employs a form of custodial transfer [11] whereby devices promise to hold copies of given objects until they can pass that responsibility on to some other device. When a device adds content to its local data store as a result of a matching placement rule, it signals its intent to hold the object via a flag in the metadata.

If placement rules later change, or the device learns of newer higher-priority data that it would prefer to hold, it issues a new metadata update removing its promise to keep the object in the future. At this point, however, the promise to hold still applies to the original data holder. Its responsibility continues to apply until some other device authors a generation that falls strictly later than the one which removed the promise, and includes a new or existing promise to hold that same data item. If two different devices holding the last two copies of an object each simultaneously announce their desire to remove it, then the generations that contain these modifications cannot be totally ordered. Neither device will be able to delete the object, as neither can identify another device that has accepted responsibility for storing the object.

This protocol ensures that, as long as no devices are lost, stolen, or broken, each non-deleted item will have at least one live replica in the device collection. This property does not depend on the existence or correctness of placement rules: applications may delete or modify placement rules without needing to ensure that some other rule continues to apply to that object.

Figure 4 shows an example content sync that continues where the metadata sync of Figure 3 leaves off. To match the user's workflow, the target device has a placement rule matching photos the camera creates; the camera has no such rule and thus tries to push its photos to other devices. When the target device receives the camera's metadata update at time t_2, it evaluates its own placement rules, and adds P_{cid} to its list of content it desires. The generation `C:1` that T received included P_{cid}, so T knows that C has a copy (the *hold* bit is set) of P_{cid} that it wants to delete (the *purge* bit). At t_3, T sends a getContent(P_{cid}) request to C, which replies with the new photo. Because T intends to keep P, it adds a *hold* bit to P_{cid} in the next generation it publishes, `T:1`.

At t_4, the devices synchronize again and the camera and target again contain identical state. But the camera now knows an important fact: the target (as of last contact) contained a copy of P, knew that C did not promise to keep P via the *purge* bit, and hence the target has accepted responsibility (*hold* but not *purge*) for storing P. Thus, at t_5, the camera can safely delete P if it needs to reclaim that space for new items, placing the system in a stable state matching the user's preferences.

4 Implementation

Eyo's prototype implementation consists of a per-user daemon that runs on each participating device and handles all external communication, and a client library that implements the *Eyo* storage API. The daemon is written in Python, runs on Linux and Mac OSX, keeps open connections (via UIA) to each peer device whenever possible, and otherwise attempts to reestablish connections when UIA informs *Eyo* that new devices are reachable. It uses SQLite [43] to hold the device's metadata store, and to implement *Eyo* queries. The daemon uses separate files in the device's local filesystem to store content, though it does not expose the location of those files to applications. The *Eyo* implementation uses XML-RPC for serializing and calling remote procedures to fetch metadata updates, and separate HTTP channels to request content objects. This distinction ensures that large content fetches do not block further metadata updates. Larger content objects can be fetched as a sequence of smaller blocks, which should permit swarming transfers as in DOT [47] or BitTorrent [6], although we have not yet implemented swarming transfers. We implemented *Eyo* API modules for Python and a library for C applications. The client libraries fulfill most application requests directly from the metadata store via SQLite methods, though they receive watch notifications from the daemon via D-Bus [8] method calls.

5 Evaluation

We explore the performance of *Eyo* by examining the following questions:

- Is *Eyo*'s storage model useful for applications and users?
- Can *Eyo* resolve conflicts without user input?
- Do *Eyo*'s design choices, such as splitting metadata from content, unduly burden devices' storage capacity and network bandwidth?
- Are *Eyo*'s continuous synchronization protocols efficient in terms of the bandwidth consumed, and the delay needed to propagate updates?

We employ three methods to evaluate *Eyo*: (1) adapting existing applications to use *Eyo*'s storage API instead of their native file-based storage to examine the modification difficulty and describe the new features of the modified versions, (2) storing example personal data collections to examine storage costs, and (3) measuring *Eyo*'s synchronization protocol bandwidth and delays to compare against existing synchronization tools.

The purpose for adapting existing applications to use *Eyo* as their primary storage interface is to examine whether *Eyo*'s API is a good match for those uses, describe how those applications use the *Eyo* API, and how difficult those changes were. We focus on two types of applications: (1) media applications, where users do not currently see a device-transparent data collection, and (2) email, where users already expect a device-transparent view, but typically only get one today while connected to a central server. We modified two media players, Rhythmbox and Quod Libet, the Rawstudio photo manager, and the gPodder podcast manager, to use *Eyo* instead of the local filesystem. We also built an IMAP-to-*Eyo* gateway to enable existing email clients to access messages stored in *Eyo*.

We evaluate *Eyo*'s storage and bandwidth costs using three data collections: email, music, and photos. These collections served as a basis for a synthetic workload used to measure bandwidth costs and storage costs due to disconnected devices.

We compare *Eyo*'s synchronization protocols to existing synchronization tools, Unison and MobileMe. Although neither tool aims to provide device-transparent access to a data collection, the comparison does verify that the performance of *Eyo*'s metadata synchronization protocol is independent of the number of objects in the collection, and demonstrates the need for direct peer-to-peer updates.

5.1 *Eyo* Application Experiences

Adapting existing applications to use *Eyo* is straightforward. Table 1 summarizes the changes made to each application. In each case, we needed to modify only a small portion of each application, indicating that adopting the *Eyo* API does not require cascading changes through the entire application. The required changes were limited to modules composing less than 11% of the total project size for the C-based applications, and significantly less for the Python applications. The C-based application changes were spread over a few months; the python applications needed only a few days of work.

***Eyo* provides device-transparency.** *Eyo* transforms the existing media applications from stand-alone applications with no concept of sharing between devices into a distributed system that presents the same collection over multiple devices. The changes do not require any user interface modifications to support device transparency; users simply see a complete set of application objects rather than the local subset. However, some user interface changes are necessary to expose placement rules and conflict resolution to the user.

Size (lines)	Rawstudio	Rhythmbox	QuodLibet	gPodder	Email
total project	59,767	102,000	16,089	8,168	3,476
module size	6,426	9,467	428	426	312
lines added	1,851	2,102	76	295	778
lines deleted	1,596	14	2	2	N/A
language	C	C	python	python	python
content	⟵ individual files ⟶				N/A
metadata	central DB, sidecar files	⟵ central DB ⟶			N/A

Table 1: Comparisons of applications adapted to *Eyo*, including lines of code changed along with descriptions of the application's original organization for storing metadata and content. For email, the "total project" size only includes Twisted's IMAP module and server example code, and "lines added" includes all of our newly written code.

Device transparency brings new features to the applications. For example, Rhythmbox and QuodLibet can show the user's entire media collection, even when content is not present, allowing users to search for items and modify playlists from any device. In Rawstudio, users can search for or organize photos in the entire collection, even when the content is missing. Surprisingly few changes were necessary to support missing content. This is because applications already have code paths for missing files or unreachable network services. Content that is not on the current device triggers these same code paths.

Users rarely encounter metadata conflicts. As a consequence of device transparency, users may encounter conflicts from concurrent changes on multiple devices. These concurrent changes result in multiple head versions of these objects when connectivity resumes. For changes to distinct pieces of metadata, the version history *Eyo* provides permits applications to resolve concurrent changes simply by applying the union of all user changes; *Eyo*'s client library makes this straightforward.

For concurrent changes to the same piece of metadata, the application must manually resolve the conflict because the correct policy depends on the application and the metadata item. In most cases, users are never aware when concurrent updates occur, as the applications perform these operations automatically. For example, if one device changes an email message status to "read" while another device changes the status to "replied", *Eyo* will signal a conflict to the application. However, the IMAP gateway knows that these updates are composable and resolves the conflict without user intervention.

Application	Type	User-Visible Conflicts Possible?	Why?
IMAP	Email Gateway	No	Boolean flag changes only
gPodder	Podcast Manager	No	User cannot edit metadata directly
Rhythmbox	Media Player	Yes	Edit Song title directly
QuodLibet	Media Player	Yes	Edit Song title directly
Rawstudio	Photo Editor	Yes	Edit settings: contrast, exposure...

Table 2: Description of whether applications can handle all version conflicts internally, or must show the presence of multiple versions as a result of some concurrent events, along with an explanation or example of why that result holds for each application.

	Email	Music	Photos
number of objects	724230	5299	72380
total content size	4.3 GB	26.0 GB	122.8 GB
native metadata size	169.3 MB	2.6 MB	22.6 MB
Eyo metadata size	529.6 MB	5.8 MB	52.9 MB
metadata/content overhead	12%	0.02%	0.04%
metadata store per object	766 bytes	1153 bytes	767 bytes

Table 3: Metadata store sizes for example datasets. The native metadata size is the size of the attribute key/value pairs before storing in *Eyo*. The *Eyo* metadata size is the on-disk size after adding all objects.

As shown in Table 2, it is possible to cause end-user visible effects. For example, Rhythmbox and QuodLibet allow users to modify metadata directly in the UI, which may require manual intervention to resolve. However, these kinds of user-visible conflicts only arise due to manual, concurrent changes and are rare in practice.

In Rawstudio, during the course of editing on two devices, users may create conflicting versions of a photo. Rather than hiding the change or requiring immediate conflict resolution, *Eyo* exposes each version as a "development version" of the photo. While this feature is typically used to let the user test different exposure and color settings, *Eyo* uses the feature to show concurrent branches of the photo.

In the other applications, gPodder and email, user-visible conflicts are impossible, as users cannot edit individual metadata tags directly. These two applications *never* show multiple versions to end users, even though the underlying system-maintained version histories exhibit forks and merges. The ability to hide these events demonstrates the usefulness of keeping system-maintained version histories so that applications face no ambiguity about the correct actions to take.

5.2 Metadata Storage Costs

To determine the expected size of metadata stores in *Eyo*, we inserted three modest personal data sets into *Eyo*: the email, music, and photo collections a single user gathered over the past decade. We included a collection of email messages as a worst-case test; this collection includes a large number of very small objects, so the metadata overhead will be much larger than for other data types. Table 3 shows the resulting metadata store sizes.

The table shows that for each of the data types, *Eyo*'s metadata store size is approximately 3 times as large as the object attributes alone. The overhead comes from database indexes and implementation-specific structures.

The most important feature this data set illustrates is that the size of the metadata store is roughly (within a small constant factor) dependent only on the number of individual objects, not the content type nor the size of content objects. The number of objects, along with the amount of metadata per object, thus provides a lower bound on the necessary storage capacity of each device.

The total metadata size in this example (less than 600 MB) is reasonable for today's current portable devices, but the total content size (153 GB) would not fit on a laptop only a few years old nor on many current portable devices. Including video content would further reduce the relative amount of overhead *Eyo* devotes to storing object metadata.

5.3 Bandwidth Costs

In addition to storage costs, the metadata-everywhere model places bandwidth costs on all devices, even when those devices do not store newly created objects.

To measure bandwidth costs, we placed a pair of object-generating devices on the same network and a remote device on a different network with a slow link to the object-generating devices. The object-generating devices create new objects at exponentially distributed times at a variable average rate, attaching four kilobytes of attributes to each new object (larger than the median email message headers considered in Section 5.2). The remote object has no placement rules matching the new objects, so it does not fetch any of the associated content. As such, all of the bandwidth used by the remote device is *Eyo* metadata and protocol overhead.

The bandwidth consumed over the slow link, as expected, relates linearly with the update rate. If the slow link had a usable capacity of 56 kbps, and new updates arrive once per minute on average, the remote device must spend approximately 1.5% of total time connected to the network in order to stay current with metadata updates. This low overhead is expected intuitively: small portable devices routinely fetch all new email messages over slow links, so the metadata bandwidth for comparable content will be similar.

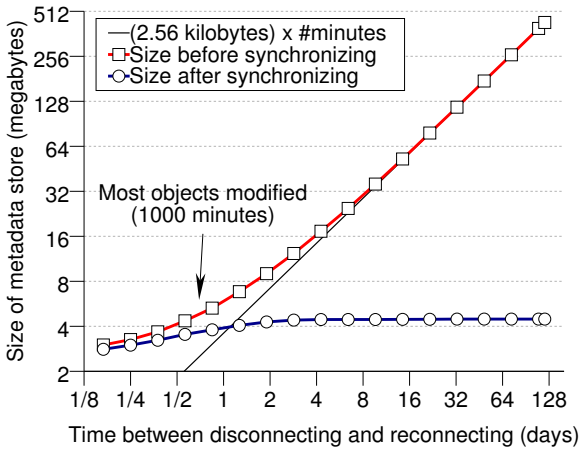


Figure 5: Storage consumed by metadata versions queued for a disconnected device (Log-Log plot).

System	Description
Unison	Delays of at least 1 second for small collections. Large collections take significantly longer: 23 seconds for an existing collection of 500K objects 87 seconds for 1M objects
MobileMe	Most updates arrive after between 5 and 15 seconds. Occasionally as long as 4 minutes. Delay does not depend on collection size.
Eyo	All delays fall between 5 and 15 milliseconds. Delay does not depend on collection size.

Table 4: Synchronization Delay Comparison: Time to propagate one new update to an existing data collection between two devices on the same local network.

5.4 Disconnected Devices

When an *Eyo* device, R, is disconnected from the rest of the group due to network partitions, or because the device in question is turned off, the other devices will keep extra metadata object versions, which might prove necessary to construct causally ordered version graphs once R returns.

In this measurement, we place an initial set of 1000 non-conflicting objects synchronized across the three devices. The remote device R then disconnects from the network, and stays disconnected for a single period of time Δt ranging from four hours to four months. Starting after R is out of communication, the other replicas generate new versions to one of the existing objects at an average rate of once per minute, attaching 2 kilobytes of unique metadata, so the devices save no space by storing only changed attributes.

After the interval Δt, we measure the size of the *Eyo* metadata store on the generating devices, allow R to reconnect and synchronize, let each device prune its metadata, and then measure the metadata store again. Figure 5 shows the before (square markers) and after (circle markers) sizes as a function of the disconnect interval Δt. The figure shows two regions, for Δt before and after 1000 minutes, the point at which most objects have been modified. For $\Delta t \gg 1000$ minutes, the system reaches a steady state where the size of the metadata store is proportional to the amount of time passed, but after returning and synchronizing shrinks to a constant size independent of the amount of time spent disconnected. The amount of recoverable storage is the difference between the two curves. The current implementation stores exactly one version beyond those strictly necessary to go back to the newest unique ancestor for each object, which is why this steady state size is larger than the initial storage size, and why the post-synchronization size changes during the initial non-steady state region.

A collection with more objects (for example, the one shown in Section 5.2) would show a much smaller fraction of recoverable storage than this example. The absolute amount of recoverable space would be the identical given the same sequence of updates.

All of the object types shown in Table 3 contain immutable contents, so disconnected devices using those data types cause overhead in *Eyo*'s metadata store, but not the content store. If updates change content as well, then the storage costs would be proportionally larger.

Figure 5 shows that a long-term uncommunicating device can cause unbounded growth of the metadata store on other devices. If this absence persists long enough that a device runs out of space, *Eyo* can present the user with two options: turn on and synchronize the missing device, or evict it from the system. Evicting the missing device, as discussed in Section 3.3, does not require a consensus vote of the remaining devices. Temporarily evicting a device allows the remaining devices to truncate history and preserve data until re-adding the missing device later.

These results show that users are unlikely to encounter problems due to accumulating metadata in practice, as large collections and infrequently used devices alone cannot cause problems. It is instead the rate of individual edits that consumes excess space. None of the applications we have examined generate changes anywhere near the frequency that this experiment assumes.

5.5 Synchronization Comparison

This section compares the latency of exchanging a single small update between two physically adjacent devices using *Eyo* to two existing classes of synchronization tools: a point-to-point file synchronizer, Unison [3], and a cloud service, MobileMe [29]. In this experiment, two devices initially hold a synchronized data collection

with some number of existing small or metadata-only objects. One device then makes a single minimal change, and we measure the time it takes for that update to appear on the second device. Table 4 summarizes the results. Since Unison is a stand-alone synchronizer, the measurement time includes the time to start up the program to send an update, which results in delays of around one second even for very small data collections. After starting, Unison first iterates over the local data collection to determine which files have changed. For large data collections, this time dominates the end-to-end delay, resulting in delays of tens of seconds for collections of a few hundred thousand individual objects.

MobileMe and *Eyo* both run continuously and continually track object changes that need propagation to other devices as applications edit data. Neither suffers a startup delay, and delays are independent of the number of objects in the collection. Although both systems send similar amounts of data (less than 10 kilobytes), MobileMe updates take between several seconds to several minutes to propagate, which is long enough for a person to notice the delay. *Eyo*'s delays in this topology fall between 5 and 15 milliseconds.

MobileMe's star topology requires that all updates pass through a distributed cloud system, even if the two devices are physically adjacent on the same local network, as in this example. *Eyo*, in contrast, discovers local network paths, and uses those to send updates directly to the local device.

Eyo can share types of data for which the other two are unsuited. Neither could store a music collection or a photo catalog shared between devices. If two devices read the catalog at startup and each later write some changes, the last write would win, and the other device's version of the file would be preserved separately but marked as a conflict. The user would have to choose one or the other versions, as the other synchronization tools provide no help for the application to resolve the concurrent changes automatically. *Eyo*, in contrast, naturally shares metadata about for these types of collections without requiring a user to routinely manage conflicts.

6 Related Work

The two systems most closely related to *Eyo* are Cimbiosys [36] and Perspective [40]. Though neither attempts to provide device transparency, *Eyo* shares ideas, like placement rules, with both. Cimbiosys provides an efficient synchronization protocol to minimize communication overhead while partially replicating objects across large groups of devices, but provides no way for a device to learn of all objects without storing all such objects. Perspective allows users to see their entire collection spanning several devices, but disconnected devices cannot continue to see the complete collection. Neither of these systems preserve object history to help applications deal with concurrent updates. Polygraph [26] discusses extensions to Cimbiosys to guard against compromised devices. *Eyo* could apply these approaches for the same purposes.

Optimistic Replication Coda [22], Ficus [20], Ivy [30], and Pangaea [39] provide optimistic replication and consistency algorithms for file systems. Coda uses a centralized set of servers with disconnected clients. Ficus and Ivy allow updates between clients, but do not support partial replicas. Pangaea handles disconnected servers, but not disconnected clients. An extension to Ficus [37] adds support for partial replicas, but removes support for arbitrary network topologies.

BlueFS [31] and EnsemBlue [33] expand on approaches explored by Coda to include per-device affinity for directory subtrees, support for removable devices, and some consideration of energy efficiency. *Eyo*'s lookup and watch notifications provide applications with similar flexibility as EnsemBlue's persistent queries without requiring that a central server know about and process queries.

Podbase [35] replicates files between personal devices automatically whenever network conditions permit, but does not provide a way to specify placement rules or merge or track concurrent updates.

Bayou [46] provides a device transparent view across multiple devices, but does not support partial replicas, and requires all applications to provide merge procedures to resolve all conflicts. Bayou requires that updates be *eventually-serializable* [12]. *Eyo* instead tracks derivation history for each individual object, forming a partial order of happened-before relationships [24].

PersonalRAID [42] tries to provide device transparency along with partial replicas. The approach taken, however, requires users to move a single portable storage token physically between devices. Only one device can thus use the data collection at a given time.

Systems like TierStore [9], WinFS [27], PRACTI [4], Pheme [49], and Mammoth [5] each support partial replicas, but limit the subsets to subtrees of a traditional hierarchical filesystems rather than the more flexible schemes in Cimbiosys, Perspective, and *Eyo*. TierStore targets Delay-Tolerant-Networking scenarios. WinFS aims to support large numbers of replicas and, like *Eyo*, limits update messages to the number of actual changes rather than the total number of objects. PRACTI provides consistency guarantees between different objects in the collection. *Eyo* does not provide any such consistency guarantees, but *Eyo* does allow applications to coherently name groups of objects through the exposed persistent object version and content identifiers.

Several of these systems make use of application-specific resolvers [38, 23], which require developers to

construct stand-alone mechanisms to interpret and resolve conflicts separately from the applications that access that data. *Eyo*'s approach instead embeds resolution logic directly into the applications, which avoids the need to recreate application context in separate resolvers. Presenting version history directly to the applications, instead of just the final state of each conflicting replica, permits applications using *Eyo*'s API to identify the changes made in each branch.

Star Topologies Many cloud-based storage systems provide a traditional filesystem API to devices (e.g., Dropbox [10], MobileMe's iDisk [21], and ZumoDrive [50]) or an application-specific front end atop one of the former systems (e.g., Amazon's Cloud Player [1]). These systems require that the central cloud service store all content in the system, and provides a filesystem API for devices. While these systems provide a centralized location for storing content, they do not enable disconnected updates between devices, or handle metadata about the objects that changes on multiple devices. Other systems such as Amazon's S3 [2], use a lower-level put-get interface, and leave all concurrency choices to the application using it. *Eyo* could use a system like S3 as one repository for object content or for metadata collection snapshots for added durability.

A number of systems build synchronization operations directly into applications so that multiple clients receive updates quickly, such as one.world [19], MobileMe [29], Live Mesh [25] , and Google Gears [17]. In these systems a centralized set of servers hold complete copies of the data collections. Applications, either running on the servers themselves or on individual clients, retrieve a subset of the content. Clients can neither share updates directly nor view complete data collections while disconnected from the central hub.

Point to point synchronization: Point-to-point synchronization protocols such as rsync [48], tra [7], and Unison [3] provide on-demand and efficient replication of directory hierarchies. None of these systems easily extend to a cluster of peer devices, handle partial replicas without extensive hand-written rules, or proactively pass updates when connectivity permits.

Attribute Naming Storage system organization based on queries or attributes rather than strict hierarchical names have been studied in several single-device settings (e.g., Semantic File Systems [15], HAC [18], and hFAD [41]) and multi-device settings (e.g., HomeViews [14]), in addition to optimistic replication systems.

7 Summary

Eyo implements the *device transparency* abstraction, which unifies collections of objects on multiple devices into a single logical collection. To do so, *Eyo* uses a novel storage API that (1) splits application-defined metadata from object content and (2) allows applications to define placement rules. In return for using the new API, *Eyo* provides applications with efficient synchronization of metadata and content over peer-to-peer links. Evaluation of several applications suggests that adopting *Eyo*'s API requires only modest changes, that most conflicting updates can be handled automatically by the applications without user intervention, and that *Eyo*'s storage and bandwidth costs are within the capabilities of typical personal devices.

Acknowledgments

We would like to thank Ansley Post, Jamey Hicks, John Ankcorn, our shepherd Ed Nightingale, and the anonymous reviewers for their helpful comments and suggestions on earlier versions of this paper.

References

[1] CloudPlayer. http://amazon.com/cloudplayer/.

[2] Amazon S3. http://aws.amazon.com/s3/.

[3] S. Balasubramanian and Benjamin C. Pierce. What is a File Synchronizer? In *Proceedings of MobiCom*, 1998.

[4] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proceedings of NSDI*, 2006.

[5] Dmitry Brodsky, Jody Pomkoski, Shihao Gong, Alex Brodsky, Michael J. Feeley, and Norman C. Hutchinson. Mammoth: A Peer-to-Peer File System. Technical Report TR-2003-11, University of British Columbia Dept of Computer Science, 2002.

[6] Bram Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[7] Russ Cox and William Josephson. File Synchronization with Vector Time Pairs. Technical Report MIT-CSAIL-TR-2005-014, MIT, 2005.

[8] D-Bus. http://dbus.freedesktop.org/.

[9] Michael Demmer, Bowei Du, and Eric Brewer. TierStore: A Distributed File-System for Challenged Networks. In *Proceedings of FAST*, 2008.

[10] Dropbox. http://dropbox.com/.

[11] Kevin Fall, Wei Hong, and Samuel Madden. Custody Transfer for Reliable Delivery in Delay Tolerant Networks. Technical Report IRB-TR-03-030, Intel Research Berkeley, 2003.

[12] Alan Fekete, David Gupta, Victor Luchangco, Nancy A. Lynch, and Alexander A. Shvartsman. Eventually-Serializable Data Services. In *Proceedings of PODC*, 1996.

[13] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent Personal Names for Globally Connected Mobile Devices. In *Proceedings of OSDI*, 2006.

[14] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy. Homeviews: Peer-to-Peer Middleware for Personal Data Sharing Applications. In *Proceedings of SIGMOD*, 2007.

[15] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and Jr. James W. O'Toole. Semantic File Systems. In *Proceedings of SOSP*, 1991.

[16] Git. http://git.or.cz/.

[17] Google Gears. http://gears.google.com.

[18] Burra Gopal and Udi Manber. Integrating Content-based Access Mechanisms with Hierarchical File Systems. In *Proceedings of OSDI*, 1999.

[19] Robert Grimm, Janet Davis, Eric Lemar, Adam Macbeth, Steven Swanson, Thomas Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. System Support for Pervasive Applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[20] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of USENIX Summer*, 1990.

[21] iDisk. http://apple.com/idisk.

[22] James J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of SOSP*, 1991.

[23] Puneet Kumar and M. Satyanarayanan. Flexible and Safe Resolution of File Conflicts. In *Proceedings of USENIX*, 1995.

[24] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[25] Live Mesh. http://www.livemesh.com.

[26] Prince Mahajan, Ramakrishna Kotla, Catherine Marshall, Venugopalan Ramasubramanian, Thomas Rodeheffer, Douglas Terry, and Ted Wobber. Effective and Efficient Compromise Recovery for Weakly Consistent Replication. In *Proceedings of EuroSys*, 2009.

[27] Dahlia Malkhi, Lev Novik, and Chris Purcell. P2P Replica Synchronization with Vector Sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

[28] John McCutchan. inotify. http://inotify.aiken.cz/.

[29] MobileMe. http://www.apple.com/mobileme/.

[30] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of OSDI*, 2002.

[31] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and Storage Flexibility in the Blue File System. In *Proceedings of OSDI*, 2004.

[32] D. Scott Parker Jr., Gerald J. Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of Mutual InConsistency in Distributed Systems. *IEEE Transactions on Software Engineering*, SE-9(3):240–247, May 1983.

[33] Daniel Peek and Jason Flinn. EnsemBlue: Integrating Distributed Storage and Consumer Electronics. In *Proceedings of OSDI*, 2006.

[34] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of SOSP*, 1997.

[35] Ansley Post, Juan Navarro, Petr Kuznetsov, and Peter Druschel. Autonomous Storage Management for Personal Devices with PodBase. In *Proceedings of Usenix ATC*, 2011.

[36] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A Platform for Content-based Partial Replication. In *Proceedings of NSDI*, 2009.

[37] David Ratner, Peter L. Reiher, Gerald J. Popek, and Richard G. Guy. Peer Replication with Selective Control. In *Proceedings of Intl. Conference on Mobile Data Access*, 1999.

[38] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of USENIX Summer*, 1994.

[39] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of OSDI*, 2002.

[40] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: Semantic Data Management for the Home. In *Proceedings of FAST*, 2009.

[41] Margo Seltzer and Nicholas Murphy. Hierarchical File Systems are Dead. In *Proceedings of HotOS*, 2009.

[42] Sumeet Sobti, Nitin Garg, Chi Zhang, Xiang Yu, Arvind Krishnamurthy, and Randolph Y. Wang. PersonalRAID: Mobile Storage for Distributed and Disconnected Computers. In *Proceedings of FAST*, 2002.

[43] http://www.sqlite.org/.

[44] Jacob Strauss, Chris Lesniewski-Laas, Justin Mazzola Paluska, Bryan Ford, Robert Morris, and Frans Kaashoek. Device Transparency: a New Model for Mobile Storage. In *Proceedings of HotStorage*, October 2009.

[45] http://subversion.tigris.org.

[46] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, and Alan J. Demers. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of SOSP*, 1995.

[47] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An Architecture for Internet Data Transfer. In *Proceedings of NSDI*, 2006.

[48] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[49] Jiandan Zheng, Nalini Belaramani, and Mike Dahlin. Pheme: Synchronizing Replicas in Diverse Environments. Technical Report TR-09-07, University of Texas at Austin, 2009.

[50] ZumoDrive. http://zumodrive.com/.

Autonomous storage management for personal devices with PodBase

Ansley Post[†] *Juan Navarro*[¶] *Petr Kuznetsov*[§] *Peter Druschel*[†]
[†]*MPI-SWS* [§] *TU Berlin/Deutsche Telekom Laboratories* [¶]*TU Munich*

Abstract

People use an increasing number of personal electronic devices like notebook computers, MP3 players and smart phones in their daily lives. Making sure that data on these devices is available where needed and backed up regularly is a time-consuming and error-prone burden on users. In this paper, we describe and evaluate PodBase, a system that automates storage management on personal devices. The system takes advantage of unused storage and incidental connectivity to propagate the system state and replicate files. PodBase ensures the durability of data despite device loss or failure; at the same time, it aims to make data available on devices where it is useful.

PodBase seeks to exploit available storage and pairwise device connections with little or no user attention. Towards this goal, it relies on a declarative specification of its replication goals and uses linear optimization to compute a replication plan that considers the current distribution of files, availability of storage, and history of device connections. Results from a user study in ten real households show that, under a wide range of conditions, PodBase transparently manages the durability and availability of data on personal devices.

1 Introduction

Modern households have multiple personal electronic devices, such as digital cameras, MP3 players, gaming devices and smart phones, in addition to desktop and notebook computers. As users increasingly depend on such devices, it is important to ensure the *durability* of data in the event of loss or failure of a device, and the *availability* of the latest data on all appropriate devices.

Ensuring that data is durable is an onerous task even for a single home computer, and the situation is getting worse as the number and diversity of devices increase. Users must keep track of all devices that need to be backed up and perform the appropriate actions on a regular basis. Anecdotal evidence suggests that many users fail to ensure the durability of their data [14, 17]. Thus, users face the risk of data loss, just as they are becoming increasingly dependent on digital information.

Making sure that a given data object is available on all the devices that need it is equally burdensome. A user must regularly connect and synchronize devices to ensure, for instance, that changes to her address book are propagated to all communication devices, and that additions to her music library are present on all devices capable of playing music.

In this paper we present *PodBase*, a system that manages data on personal devices in an autonomous, decentralized, device- and operating system-independent manner. The system is transparent to the user, takes advantage of unused storage space and exploits incidental pairwise connectivity that naturally occurs among the devices, (e.g., via Wi-fi, Bluetooth or USB).

With PodBase, each device stores metadata that describes a household's devices and data. During pairwise connections, devices reconcile their metadata and exchange data. Over time, metadata and data propagate among a household's devices. PodBase progresses toward a state where, subject to available storage and in order of decreasing priority, (i) the contents of any failed device are restored to a replacement device, (ii) each object has a certain minimal number of replicas, and (iii) each object is available on devices that can potentially use it.

Results from our user study show that many households have sufficient storage and connectivity to permit full replication. However, there is typically not one hub device with plenty of storage to which all other devices are regularly connected with sufficient bandwidth. To ensure full and timely replication, PodBase must therefore be able to use free space on all devices, replicate data between any pair of devices, and possibly even move data via sequential pairwise connections.

Given the vast space of possible configurations, device connection sequences and replication plans, design-

ing an appropriate replication algorithm for PodBase is not straightforward. Simple, greedy algorithms are stable and robust but tend to get stuck in local minima. PodBase instead uses linear optimization to compute an adaptive replication plan from a declarative specification of the goal state, and a local view of the current replication, available storage and history of device connections. As a result, PodBase is highly adaptive and provably stable. Moreover, it finds sophisticated solutions in unexpected scenarios. For instance, without being programmed for this case, the system takes advantage of "sneakernets", i.e. mobile devices, to transport data between home and office, thus avoiding slow broadband connections.

The rest of this paper is structured as follows. Section 2 states the requirements. We discuss related work in Section 3. Section 4 presents the design of PodBase and Section 5 describes its replication algorithm. Section 6 presents our evaluation and Section 7 concludes.

2 Requirements

PodBase is intended for a household with one or more users and a set of shared personal devices. Based on the results of a feasibility study [20], we can characterize this environment as follows:

- Devices are periodically connected, such that any pair of devices can eventually communicate via a series of sequential pairwise connections.
- A device may fail or be lost at any time. However, the failure or loss of many devices during a short period of time is unlikely.
- Devices may be turned off when not in use; it cannot be assumed that any one device is always online.
- The system must be able to handle a wide range of usage patterns and device configurations, without attention from an expert system administrator.

An important aspect of the target environment is that most users don't have the expertise, interest or time to manage data and storage on their devices. They expect the system to do something reasonable automatically. Unlike a system designed for expert users (like the authors and readers of this paper), PodBase must be able to achieve its goals with little user expertise and attention.

2.1 Desired system behavior

In this section, we describe the desired system behavior intuitively and by example. A more detailed description of PodBase's properties, design and implementation follows in subsequent sections.

PodBase aims to relieve users from having to worry about the *durability* and *availability* of their data. Durability requires that the failure or loss of a device not result in the loss of user data. Availability requires that each device store the latest collection of data *relevant* to that device. For example, each communication device should store the latest version of the address book and, subject to available storage space, a shared music collection should be available on all devices capable of playing music.

As an example, Alice and Bob share a household. Alice has a notebook, an MP3 player and an external USB hard drive. Bob has a notebook and a desktop computer at his office. Their home has a wireless network connected to the Internet via a broadband connection. On workdays Alice and Bob bring their notebooks to their offices and perform their daily work, such as writing documents and using email.

At night both return home with their notebooks and use them to surf the web, play games, or listen to music. Although they have important data stored on their notebooks, they rarely back up their data.

PodBase should automatically perform the following tasks without any explicit action by Alice or Bob:

- Every night, new or modified files are replicated, in cryptographically sealed form, between Alice and Bob's notebooks via the wireless network. (This works even when they are on vacation, e.g., when the pictures Alice uploads from her camera are replicated on Bob's notebook.)
- When Bob purchases a new CD and rips it to his hard drive, a replica of the mp3 file is later moved to Alice's notebook. When Alice connects her MP3 player to charge, it also receives the new music.
- Whenever Alice or Bob edit their personal address books, the changes are automatically propagated to their other communication devices.
- Whenever Alice's USB hard drive is connected to her laptop, additional replicas of the files and replicas on her laptop are made.
- Bob's office desktop is connected to his home via a broadband connection. Rather than transfer data using the slow connection, the system uses Bob's notebook disk to rapidly replicate data between home and work.
- When Bob's notebook is running low on disk space (after removing any replicas), the system asks Bob if it should move not recently accessed movie files to Alice's USB drive, which has plenty of space.

PodBase can recover from otherwise costly incidents. For example, imagine Alice's laptop is stolen. With PodBase, she is able to restore the data on the lost device's hard drive to her replacement notebook. When she connects over the wireless network to Bob's notebook, some files from her stolen notebook are restored on the new device. When she later connects her new notebook to the USB drive, the remaining files are restored. Thanks to the replication between home and Bob's office, they could recover all data even after a total loss of the home or office devices.

An important goal we set ourselves for PodBase is transparency: the system's background activity should not affect users' experience during normal operation. By default, the system does not remove user files, automatically propagate changes to user files or attempt to reconcile conflicting versions of concurrently modified files. Instead, PodBase maintains all versions of a file along with their modification history. Optional plug-ins can define file type-, device-, or application-specific consistency semantics.

PodBase's transparency is consistent with the principle of least surprise: by default, the installation of the system should not change a device's user-visible behavior during normal operation. Advanced behavior (e.g., automatic propagation of changes to the address book) can be enabled explicitly by enabling appropriate plug-ins.

3 Related work

PodBase is in the spirit of Weiser's Ubiquitous Computing vision [39], as it transparently manages storage on personal devices. To the best of our knowledge, no prior system provides automatic durability and availability of data on personal devices, without relying on central storage, a fast Internet connection or explicit user attention.

With Personal Server [37], users carry a personal storage device and use input/output devices found in the environment. In Omnistore [10], data is maintained on a central store, while other devices interact to cache data or relay data to the store. The Roma system [32] provides a shared, centralized metadata service that can be used to build higher-level services for synchronization, consistency and availability. Apple TimeMachine [34] and Windows Home Server [40] provide automatic backup to a dedicated storage node. Unlike PodBase, the above systems rely on a dedicated storage device, are vulnerable to the failure or loss of that device, and cannot exploit unused storage on other devices.

Availability of data on a set of devices can be provided by a distributed file system that supports disconnected operation, like Bayou [33], Ficus [19], and Coda [12]. Some systems additionally support partial replication to meet the needs of mobile devices, e.g. PRACTI [1], WinFS [42], Roam [24], Ensemblue [18], the Few File system [22] and Segank [29]. Oasis [25] is an SQL-based data management system for pervasive computing applications. PodBase differs from these systems in that it replicates data for availability *and* durability, is fully automatic, takes advantage of pairwise connections and unused storage efficiently, requires no centralized server, and is device, vendor and OS-independent.

Cimbiosys [23] is a platform for content-based partial replication. Like PodBase, Cimbiosys carefully manages the amount of information that has to be exchanged during pair-wise connections. The goal of Cimbiosys is to facilitate replication by propagating updates between peer devices. Applications or users are expected to specify filters for what each device should store. Unlike PodBase, Cimbiosys does not specify a replication policy for either availability or durability, and instead provides a replication platform for higher level applications. For replication to eventually reach the desired state, Cimbiosys assumes that all devices that replicate a given collection of objects form a tree, such that a parent stores a superset of the objects stored by its children and children regularly connect to their ancestors. PodBase, on the other hand, achieves eventual consistency as long as any two devices are repeatedly connected via a sequence of pairwise connections.

Like PodBase, Perspective [27] supports automatic partial replication among mobile devices, without relying on a centralized server. However, Perspective assumes that a view is defined for each device, which specifies the set of files that should be present on the device. Files are then replicated along sequences of pairwise connections, where a file must be contained in the view of each device that appears on the path. PodBase, on the other hand, uses multi-step replication plans, where files can be placed on intermediate devices solely for the purpose of transporting them to another device. PodBase computes a replication plan automatically and dynamically to maximize durability and availability given the available free space on devices, without requiring the specification of per-device views.

One could try to simulate the effect of PodBase's replication policy in Perspective by specifying that each device's view include all files. Perspective would then replicate all files greedily as device connections occur, until each device either replicates all files or its space is exhausted. Unless most devices have enough space to store most files, however, this would likely lead to uneven replication levels and poor availability. Finally, PodBase was evaluated using an actual user deployment.

Device Transparency [30] is a storage model for mobile devices, where each device maintains global metadata. PodBase uses a similar capability as a building

block to support transparent data replication for availability and durability. Moreover, PodBase can also support devices too small to store metadata for all objects in the system.

Synchronization tools like Unison [36] synchronize data among devices, and attempt to reconcile replicas that have diverged due to concurrent edits. Windows Live Sync [41] and Live Mesh [13] allow users to sync folders on their machines. File synchronization tools like these can be used as a plug-in for PodBase. Groove [9] provides a collaborative workspace that propagates file edits automatically among a group of users. PodBase is also concerned with durability and focuses on intermittently connected devices in the home.

Pastiche [4] and FriendStore [35] implement cooperative backup on users' machines in a peer-to-peer network. PodBase replicates data for availability and durability, within a household, on intermittently connected devices, and without relying on third-party storage.

Cloud storage services (e.g. [5,15,28,31]) provide automatic backup or synchronization for mobile devices at a charge. PodBase is free, can replicate much faster because it is not limited by the upstream bandwidth of a broadband connection, exploits unused storage on existing devices, replicates among devices that are away from home (e.g. on vacation), and avoids the dependence on a single provider for data protection. Nevertheless, PodBase can take advantage of a Cloud storage service to maintain additional off-site replicas for added safelty.

Keeton et al. [11] advocate the use of operations research techniques in the design and implementation of systems. PodBase is an example of a system that uses linear optimization to adapt to its environment. Other examples include Rhizoma [43] and Sophia [38], which use logic programming to optimize cloud computing and network testbed environments, respectively. Pandora [2] uses linear optimization to optimize bulk data transfers for cost and timeliness, using a combination of Internet data transfers and the shipping of storage devices.

Since PodBase shares data among a set of intermittently connected devices, it implements a form of delay tolerant network (DTN) [6]. PodBase can be viewed as a data management application on top of a specialized DTN. The Unmanaged Internet Architecture [8] (UIA) provides zero-configuration naming and routing for personal devices. PodBase addresses the complementary problem of data management for personal devices.

A prior workshop paper [20] sketches a preliminary design of PodBase and presents results from a trace-based feasibility study. This paper contributes a revised design, a full implementation, a new replication algorithm, support for space-constrained devices, a plug-in architecture to add file type and device specific behavior, an extensive evaluation and a user study.

4 PodBase design

We start with an overview of PodBase, its user interface, operation, plug-in architecture and security aspects.

4.1 Overview

PodBase is implemented as a user level program. It keeps track of user data at the granularity of files. PodBase is oblivious to file and device types. However, PodBase supports a plug-in architecture, by which file type and device specific data management policies can be added.

PodBase distinguishes between *active devices* and *storage devices*. Storage devices include hard drives, media players and simple mobile phones. Active devices run the PodBase software and provide a user interface. An active device contains at least one storage device; additional storage devices can be connected internally or via Bluetooth or USB. The set of devices in a household form a PodBase *pool*. In each pool, there must be at least one active device, which runs the PodBase software.

Active devices communicate via the network and handle the exchange of data. Whenever two active devices communicate, a storage device is attached to an active device, or two storage devices are attached to the same active device, we say that these devices are connected. Data propagates during these pair-wise connections.

There are three different types of data on each storage device: (1) regular *user data*, (2) PodBase *file replicas*, and (3) PodBase *metadata*. Although logically separate, all of these data are stored in the device's existing file system. The PodBase replicas and metadata are cryptographically sealed and stored under a single directory.

Metadata describes a device's most recent view of the pool's state. Included in the metadata is the set of known devices and their capacities, a logical clock for each storage device and a list of all user files that PodBase manages, along with their replication state. Capacity constrained devices may store only a subset of the system's metadata, as described in Section 4.3.2.

Some of the space on a device not occupied by user data or metadata is used to replicate files for durability and availability. User data has priority over replicas. PodBase continuously monitors its storage use and seeks to keep a proportion f_{min} of the device's capacity free at all times.

When a file is modified by an application or the user, PodBase creates a new version of the file and replicates both the old and new version independently. Plug-ins (see Section 4.4) can be used to automatically apply consistent file updates, reconcile conflicting versions or purge obsolete versions in a file type-specific manner. Users can manually retrieve copies of old versions or even deleted files.

4.2 User interaction

Next, we describe how users typically interact with PodBase. Though PodBase is designed to minimize user involvement, some interaction is required. Moreover, interested, tech-savvy users have the option to change its policies.

Device Registration. When a new device is connected for the first time, PodBase asks the user if the device should be added to the storage pool.

Device Deregistration. A storage device may permanently disappear due to loss, permanent failure or replacement. If a device has not been connected for an extended period (e.g., a month), PodBase prompts the user to connect the device or else deregister it.

Data Recovery. When a storage device fails, PodBase can recover the files it stored. The user informs PodBase that she wishes to recover the data from a particular lost device onto a replacement device or onto an existing device. The PodBase software on the recovery device then obtains copies of the appropriate files during each connection.

Externalization. By default, users and applications cannot directly access replicas stored on a device. However, users with the appropriate credentials can *externalize* replicas, that is decrypt and move the cleartext of a replica into the user file portion of the device. Alternatively, externalization can be automated using a plug-in.

Warnings. PodBase warns the user when it is unable to replicate files because there is insufficient storage space or connectivity, with specific instructions to buy an additional disk or connect certain devices.

4.3 Device interaction

When two devices are connected, they reconcile their view of the system and exchange data. First, the devices reconcile their metadata. Then, PodBase determines if any of the replicas on either device should be moved, copied or deleted. Next, we detail these steps.

4.3.1 Metadata contents

The metadata consists of the following items (their purpose will become clear in the subsequent discussion):

1. Vector Clock: A vector clock, consisting of the most recent known logical clock values for each device in the pool. A device's logical clock is incremented upon each metadata change. When a device is removed from the system, its logical clock is set to a special tombstone value. Also, the metadata includes the most recently observed vector clock of each device in the storage pool.

2. Connection History: A list of the past 100 connections that have been observed between each pair of devices, their time, duration, their average and maximum throughput, as well as the network addresses used by the devices.

3. Policies: The current policy settings. Policies can be modified by sophisticated users. Installed plug-ins (Section 4.4) can also modify the policies.

Items 1–3 are included in the metadata of all devices.

4. Set of user files: Keeps track of the user files stored on each device in the pool. The content hash value, size and last modification time are recorded for each unique file. In addition, the content hashes of the last v ($v = 10$ by default) versions of each file are included (modification history).

5. Set of replicas: Keeps track of the replicas stored on each device in the pool. For each replica, its size, content hash value, and replica id are recorded.

6. Reverse map of unique files in the pool: Maps a content hash value to the set of files whose content matches the value. This mapping is used to determine the current replication level for each unique data file, considering that different files may have identical content. (PodBase de-duplicates files prior to replication.)

Each record in items 4–6 contains a version number, which corresponds to the device's logical clock at the time when the record was last modified. A small device may include only a subset of the records in items 4–6.

4.3.2 Metadata reconciliation

Metadata reconciliation is straightforward in the common case when two devices that carry the full metadata are connected. They compare their vector clocks to determine which has the more recent metadata for each device in the pool. For each such device, the more recent metadata is then merged into the reconciled metadata.

PodBase also supports devices too small to hold the full metadata. (In practice, devices smaller than about 100 MB are excluded. This is a mild limitation, since smaller storage devices are already rare at the time of this writing.) Such devices hold the full metadata for the files and replicas they store, plus some amount of partial metadata about other devices.

PodBase ensures progress and eventual consistency of metadata, even if some devices are only ever sequentially connected via small devices. To this end, PodBase seeks to place on small devices metadata that are needed to update other devices. For this purpose, it checks the last known vector clocks of all devices. PodBase selects partial metadata subject to the available space on the small

A second preimage resistant hash function is used

device, while ensuring that (i) metadata needed by more devices are more likely to be chosen, and (ii) a roughly equal number of metadata items are included for each device that the small device may encounter. This policy seeks to maximize the spread of useful information and ensure convergence of device metadata even in extreme situations where different sets of devices are connected only via a small device.

When reconciling any device L with a small device S, PodBase checks if the metadata on S can be used to update L. For a given device d whose partial metadata appears in a small device, all metadata are included that have changed within some range of versions $i < j$ of d's metadata. This metadata can be used to update L if L's current metadata version for d is at least i and less than j. If so, PodBase merges the metadata about d from S into L's metadata.

4.3.3 Replication

Once the metadata is reconciled, PodBase determines the actions, if any, that should be performed on the data. PodBase may *copy* a replica of a file, in which case the file is stored on the target device with a new random replica id (used to distinguish between replicas), while the original replica remains on the source device. A device may also *move* a replica, in which case the replica is stored on the target device with the same replica id and then deleted from the source device. Finally, a device may *delete* a replica, to make room for another replica that it believes is more important. During replication, data is transmitted in a cryptographically sealed form, and a hash of each replica's content is attached to ensure data integrity. How PodBase determines the actions that should be performed is described in Section 5.

4.3.4 Data recovery

After a device loss or failure, data can be recovered onto a replacement device at users' request. During each connection to another device, the replacement device restores as many files as possible, guided by the reconciled metadata. The most recent available version of each file is restored. Users can speed up the recovery process by connecting appropriate devices under the guidance of PodBase. The restoration is complete when the replacement device has received, directly or indirectly, from each device in the pool a metadata update no older than the time at which the lost device went out of service, and the reconciled metadata indicates that all files were restored.

4.3.5 Replica deletion

PodBase removes replicas when the free space on a device falls below f_{min}, the minimal proportion of a device's storage that PodBase keeps available at all times (by default, $f_{min} = .15$). When PodBase frees space, it considers the most replicated files first. Among files with the same replication level, PodBase first deletes replicas that have the lowest (randomly assigned) replica id among the replicas of a file, then the second lowest id, and so on. This policy ensures that different devices delete replicas of the same file only when a shortage of space dictates it, but never as a result of inconsistent metadata in partioned sets of devices. (PodBase never deletes the original or any externalized replica.)

4.4 Plug-ins

Plug-ins can be used to implement policies and mechanisms that are specific to particular file types, collections of files, device types or specific devices. Following are some example plug-ins.

Consistency: PodBase replicates each version of a file independently. A plug-in can be used to automatically propagate changes or reconcile concurrent modifications under a given consistency policy. There is a large body of work on consistency, and powerful tools exist for reconciling specific file types, e.g. [7,16,26]. Such tools can be integrated as plug-ins in PodBase.

Unified Namespace: By default, PodBase does not automatically externalize replicas. A plug-in could export files as part of a global uniform namespace on all devices. This would allow users to browse the contents of all devices, and access files available locally (subject to user access control restrictions). In combination with a plug-in that provides consistency, this would provide a simple distributed file system.

Digital Rights Management (DRM): Media files stored on a user's devices may be protected by copyright. Usually, copyright regulations allows users to maintain copies on several of their personal devices. However, if restrictions apply, then the policies appropriate for a given media type can be implemented as a plug-in.

Archiving: A plug-in can automatically watch for large, rarely accessed user files (e.g. movies). If such files occupy space on a device that is nearing capacity, the plug-in suggests moving the collection to a different device with sufficient space. If the user approves, PodBase automatically moves the files.

Content-specific policy: A content-specific plug-in can, for example, replicate and automatically externalize mp3 files on devices capable of playing music. Moreover, the plug-in can select a subset of the music collection for

placement on small devices. For instance, when replicating music on a device with limited space, a plug-in may select the most recently added music, the most frequently played music, and a random sample of other music.

As a proof of concept, we developed a plug-in that automatically externalizes replicas of mp3 files and imports them into iTunes. The plugin required around 100 lines of Java code, and two simple OS specific AppleScript scripts to interact with iTunes.

4.5 Security

PodBase uses authenticated and secure channels for all communication among devices within a pool. When a device is introduced to a PodBase pool, it receives appropriate key material to enable it to participate. Users have to present a password when they wish to interact with PodBase. Metadata and replicas are stored in cryptographically sealed form when stored on devices, in order to minimize the risk of exposing confidential data when a device is stolen. PodBase respects the file access permissions of user files – encrypted replicas can be externalized only by a user with the appropriate permissions on the file. By default, PodBase manages all of a device's contents; it can be configured to manage only specific subtrees in the namespace of a device.

The strength of PodBase's access control within a household is designed to be at least as strong as the access control between different users on the same computer. If stronger security isolation is required between devices or users, then they should not join the same pool. For instance, if a user's office computer contains confidential material that must not leave company premises, then it must not join the user's home PodBase pool.

5 Replication

We considered a number of replication algorithms. Greedy algorithms place under-replicated files on the first connected device that has space. These algorithms are simple, stable, and replicate files at the first opportunity, which is good. Unfortunately, the initial placement of a file is often sub-optimal and cannot be changed. (By definition, greedy algorithms never reconsider an earlier choice and cannot move replicas if a better placement turns out to be possible in the future.) A more sophisticated class of algorithm seeks to equalize the storage utilization of connected devices, thereby moving replicas toward devices that have space. Unfortunately, these algorithms cannot take advantage of a "shuttle device" to transport data between clusters of devices, e.g., home and office.

Extending the algorithms to cover these and other important cases while avoiding degenerate performance in unexpected cases seemed daunting. Instead, we decided to pose optimal replication declaratively as a linear optimization problem. This approach minimizes design time assumptions about system configurations and usage patterns, computes optimal solutions to unexpected cases at runtime, and has provable stability properties.

Whenever two devices connect, PodBase uses an LP solver to compute a multi-step replication plan that moves the system toward the goal state. The plan considers the current system state and likely future device connections, and specifies which replicas should be deleted, copied or moved during each connection accordingly.

In general, only the first step of the replication plan is relevant, as it concerns the currently connected devices. The subsequent steps are speculative, since they depend on which future device connections actually occur. If the actual device connected next differs from the current plan, a new plan is computed. The following subsections describe the approach in more detail. Additional detail about the LP problem formulation can be found in a technical report [21].

5.1 Replication objective

First, we wish to guarantee that files are evenly replicated on as many devices as the available space allows. As a secondary goal, we want to maximize availability by placing copies of each file on devices where it is potentially useful. In the rest of this section we define these two properties more formally.

Let D be the set of participating devices and let F be the set of files that are managed by the system. For each device $d \in D$, let $space_d$ denote its capacity, i.e., the amount of space available at d for storage of both user and replica files. For a set of files $S \subseteq F$, $size(S)$ denotes the amount of storage required to keep a copy of S. For each device d, the set of *user files* stored in that device is denoted by $\mathsf{user\text{-}files}(d)$. In particular, for each device d, $size(\mathsf{user\text{-}files}(d)) \leq space_d$.

The goal of a storage management system is to determine and maintain, for each device d, a suitable selection of files, $\mathsf{store\text{-}files}(d) \subseteq F$, to be stored on it. Files are replicated when they are selected for storage at several different devices. Moreover, at any time, such a selection must satisfy

- $\mathsf{user\text{-}files}(d) \subseteq \mathsf{store\text{-}files}(d)$, user files are never moved or deleted from devices;
- $size(\mathsf{store\text{-}files}(d)) \leq space_d$, the files stored on a device may not exceed its capacity.

Given a particular store-files selection, we say that its *replication factor* is the number of copies k of the least

replicated file in the system. More formally,

$$k = \min_{f \in F} |\{d \in D : f \in \text{store-files}(d)\}| \ .$$

Moreover, we say that the replication factor is *optimal* if there is no other file selection store-files′ with a higher replication factor.

In order to model availability, plug-ins have the option to provide an *availability selection* that assigns to each device $d \in D$ a set of files like-files(d) that it should *preferably* store. The *availability score*, or *av-score*, of a file selection store-files is then defined as the number of file copies that match the preference expressed by like-files, i.e.,

$$\textit{av-score} = \sum_{d \in D} |\text{like-files}(d) \cap \text{store-files}(d)| \ .$$

In a desired *goal state*, PodBase places at each device $d \in D$, a set store-files(d) of replicas such that the following properties are satisfied:

Durability. The replication factor is optimal, i.e., files are maximally replicated on the existing devices.

Availability. Among the file selections with optimal replication factor, store-files has a maximal *av-score*; i.e., files are replicated in devices where they are useful.

5.2 Problem formulation

The system state, the effects of the actions, as well as the objectives are modeled as a set of linear arithmetic constraints. Care must be taken to ensure the problem formulation scales. To make the optimization problem tractable, we group files into equivalence classes called categories. All files that are stored on the same set of devices are in the same category. The system state is then encoded by specifying, for each category, the total amount of space occupied by all files in that category. This significantly reduces the number of variables in the problem formulation, which no longer depends on the number of files but on the number of devices in a pool, without any loss of accuracy.

To model the connectivity among devices, a graph is constructed with a link between each pair of devices that can potentially be connected. The link weight specifies the estimated *cost* of data transfer among the devices. This cost is calculated based on the maximum connection speed and the probability that the devices will be connected on a given day, based on the history of past connections. In this calculation, more recent connections are weighted more heavily; individual measurements are filtered appropriately to reduce noise [21].

Finally, we model the actions (copy, move, delete) PodBase can perform, and their effects on the system state. In general, a sequence of connections may be required in order to affect a certain state change (e.g., copy some files from A to B, and then from B to C). The formulation then encodes how the possible sequences of actions modify the number of bytes in each category.

Encoding the problem this way enables us to symbolically describe all the possible plans that PodBase could execute in order to manipulate the distribution of files. Given this formalization, the goal is to find a plan that optimizes the desired goals.

The optimization involves multiple stages, narrowing the set of candidate replication plans in each. First, the maximal replication factor k is computed based on the available space in the system. Then, we optimize for durability by computing replication plans that can achieve a k-replication for all files. Next, we optimize for cost by narrowing the set of plans to those that minimize the sum of the link weights. In the next stage, we select among the remaining plans those that maximize availability. In the final stage, we select a plan that minimizes the number of necessary replication steps. PodBase then executes the first step of the resulting replication plan, by copying, moving or deleting replicas on the currently connected devices. For efficiency, we do not consider plans with more than three replication steps. (Few interesting plans with more steps occur in practice.)

The optimization favors cost over availability, because high cost plans are highly undesirable: they may rely on links with low bandwidth or rare connectivity. Notably, this choice still permits good availability, because the cost optimization generally leaves many candidate plans from which the availability optimization can select. The reason is that all plans involving the same set of connections have the same cost, and there is a combinatorially large number of such plans, corresponding to the different placements of replicas that can occur as a result of these connections.

The cost optimization does, however, eliminate plans that create more than k replicas, even if availability calls for more. To enable additional replication for availability, PodBase changes the order of optimizations once the durability goal has been achieved. In this case, availability is optimized before cost.

In a final step, the categories are mapped back onto individual files. In cases where the solution would require a file to be split by an action, file integrity can be fed back into the optimizer as an additional constraint. The replication process is guaranteed to converge in a bounded number of steps after the set of primary data files stabilizes.

Additional parameters could be added to the optimization by the system designer. For example, if device reliability data is available, this information can be considered by modeling a replica stored on a less reliable device

as contributing less to the durability of the associated file than a replica stored on a more reliable device. In general, such extensions are straightforward to implement. However, they do require some expertise with linear optimization to make sure the additional inputs or constraints do not cause an blow-up in the complexity and runtime of the optimization.

6 Experimental evaluation

Next, we present experimental results obtained with a prototype implementation of PodBase. We sketch the implementation, report on its overheads and verify that the system behaves as expected. Then we present measured results from a user study. Additional results, including a comparison with a simple greedy replication algorithm, are presented in the technical report [21].

6.1 Implementation

PodBase is implemented as a user-level program written in Java. Most of the code (48,512 lines) is platform-independent, with the exception of a small amount (about 1000 lines) of custom code for each supported platform (Windows 2000 and higher, Mac OS X). The platform-specific code deals with mounting disks and naming files. The implementation currently requires that storage devices export a file system interface, and that active devices are able to run Java 1.5 bytecode.

Running PodBase on platforms like cell phones or game consoles is feasible, but requires additional engineering effort. We feel that our prototype strikes a reasonable trade-off between engineering effort and research goals, because it can use the majority of devices in our study.

In our deployment, active devices contact a server (2.6Ghz AMD Opteron) running CPLEX 11.2.1 (a commercial LP solver) to compute replication plans. Using the server simplifies the installation of PodBase and is not fundamental to the system. With an additional installation step, PodBase can be configured with a local solver, like the free LP solver package clp [3].

PodBase rate-limits network and disk I/O, marks I/O as non-cacheable and runs single-threaded to avoid competing with other applications for resources. To the extent possible, we tried to ensure that users did not notice that PodBase was running in the background.

6.2 Computation and storage overhead

PodBase periodically crawls file systems to monitor the state of files. Each time a new file is discovered or an existing file is modified, the file is hashed and added to the pool's metadata. We measured the amount of time the first crawl took when a new drive was added to the system. The measurements were taken on a 2.4 GHz Apple MacBook Pro, running OS X, one author's primary computing device. The internal notebook disk contained 165,105 files with a total size of 87.4GB. The initial crawl took approximately 5 hours to complete. Subsequent crawls, which only re-compute hashes for new or modified files, took on the order of 10 minutes. (Both OS X and Windows support APIs that notify applications of any folder or file modifications. Using these APIs can dramatically reduce the need for crawling, but our implementation did not use them.)

The size of the system's metadata grows proportionally with the number of files and replicas managed by a PodBase pool. In our user study, the uncompressed metadata size ranged from 270MB to 2.5GB. This amounts to only a small fraction of the capacity of most modern storage devices. For the devices in our user study, storing the full metadata was possible in all cases. However, smaller storage devices (e.g. older USB sticks or cameras) are supported via the partial metadata mechanism.

Using the LP solver to compute a replication plan takes between one and thirty seconds for most households, and 180 seconds for the largest household in the user study. When two devices connect, replication starts immediately on a speculative basis, while the optimization runs in the background. For instance, PodBase starts to replicate greedily those files that appear the most underreplicated or that should reside on one of the devices for availability, according to the reconciled metadata. This replication can later be (partly) undone, in the case that some of it is inconsistent with the computed plan.

6.3 Data restoration

Next, we test PodBase's ability to successfully restore the contents of a lost device. We simulated the loss of a notebook after the replication phase was completed. PodBase successfully restored to a USB hard drive all 211206 files (75GB) that were present at the time of the last crawl of the "lost" notebook. The restoration took 5 hours 27 minutes to complete, which includes decrypting the replicas.

6.4 Partial metadata reconciliation

Next, we experiment with small devices that carry partial metadata. In our example, there are three devices: two full metadata devices, which never directly connect to each other; and a small device, which is connected to each of the other devices once per day. The small device is able to carry 100MB of metadata about other devices, and unable to carry actual data. The total metadata size is 2GB.

Initially, the large devices were completely unaware of each other. No new data was added after the experiment began. It took ten days or 21 connections for the metadata on the two large devices to converge, which is expected based on the relative size of the metadata and the small device. This example shows that metadata converges even in extremely constrained cases. In our experience, most devices are larger and connectivity tends to be much richer in practice, leading to much faster convergence.

6.5 User study

To study how PodBase performs in a real deployment, we asked ten members of our institute to deploy the system in their households and collected trace data over a period of approximately one month. We asked the users to, as much as possible, ignore the presence of PodBase and use their devices the way they would normally use them. Three users were given an external one terabyte USB disk, because they had insufficient free space to allow their files to be replicated.

For practical reasons, the number of households and users in our study is limited and covers a relatively short period of time. Moreover, at least one member of each household was a computer science researcher. Therefore, there is a likely bias towards users who have an interest in technology. As a result, our results may not be representative of a larger and more diverse user community, or a long-term deployment. Nevertheless, we feel that the study was tremendously valuable in identifying the difficult issues, in building our confidence that the system is feasible and addresses a real need, and in understanding the system's performance in practice.

The system was deployed and actively used over the course of two years. The data collected for the results presented in this paper were collected between July and September 2009. During this period, we collected anonymized data about file creation, modification and deletion on each device, when and where replicas were created, and which devices were connected at what times. We use these logs to generate the graphs used in the rest of this section.

First, we provide a brief overview of the households used in our deployment and the characteristics of the devices used in each.

Figure 1 shows the number of storage and active devices in each household. The number of active devices ranged from one to seven. Some households had no additional storage devices, while others had up to three. Households 1, 4 and 5 received an additional one terabyte USB disk, which is reflected in the data. Household 4 has a virtual device that is backed by 10GB .Mac cloud storage. PodBase uses this device like any other, considering its capacity and connection bandwidth.

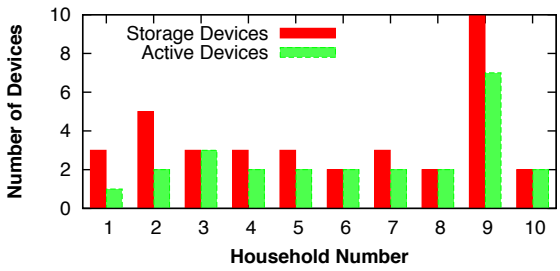


Figure 1: Number and type of devices, by household.

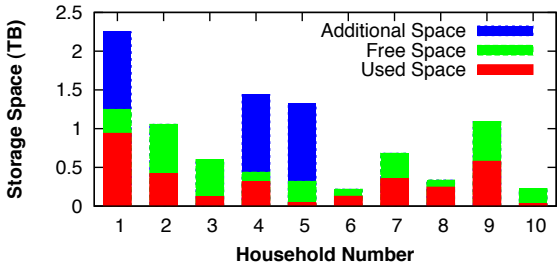


Figure 2: Storage capacity and free space on devices before PodBase begins replication. Additional space corresponds to the USB disks given to households 1, 4, 5.

Figure 2 depicts, for each household, the total size of the household's storage pool, divided into used storage and available storage at the beginning of the deployment and before PodBase was activated. The additional storage given to households 1, 4 and 5 is shown as "additional space". After this addition, seven of the households had at least half of their total storage capacity available. This does not imply that the remaining three households cannot replicate their data; whether they can depends on how much duplication there is among their existing user files.

6.5.1 Replication results

In this section, we evaluate the performance of PodBase by looking at the replication state at the beginning and the end of the (one month long) trace collection.

Let us look at the replication state of the system before the households ran PodBase. As shown in Figure 3 (left bars), many households had files that existed on only one device, leaving these files vulnerable to data loss if the device were to fail. Also, many households had a significant number of files already replicated, either as copies of the same file or different files with identical content.

The right bar in Figure 3 shows the replication state at the end of the trace collection. Five households (1–

The result for household 7 was obtained by re-playing the trace,

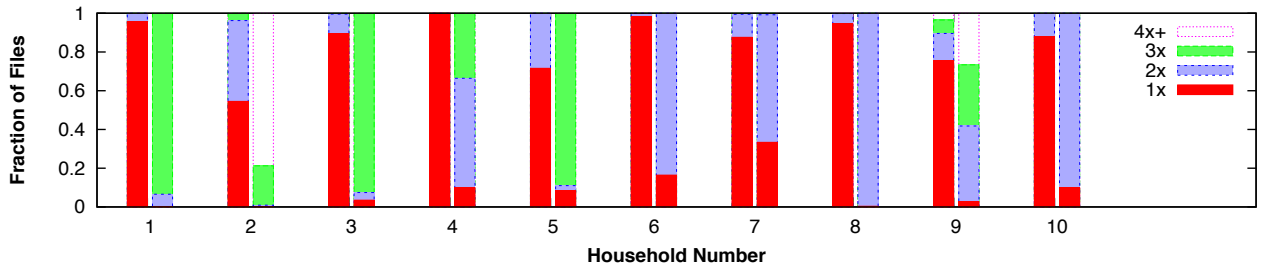


Figure 3: The initial (left bar) and final (right bar) replication status of each household.

3, 8–9) had most (more than 97%) of their files replicated. With the exception of household 9, which had not quite finished replicating its original files, the remaining households' unreplicated files were recently created or modified and had not yet been replicated at the end of the trace. Households 4, 5, and 7 were not able to replicate as much, as these households had only intermittent connectivity between a pair of their devices. These households each had two well-connected devices and one device that was either mostly offline or connected via a slow DSL connection. In these cases, all of the data was replicated between the well connected devices, but the data on the poorly connected device was not replicated fully.

Households 6 and 10 did not have enough space to replicate the remaining 19% and 10% of their files, respectively. In order to improve upon these results, the users would have had to purchase inexpensive additional storage. As a sanity check we had users from households 4 and 10 bring in their notebooks in order to confirm the diagnosis described above. Simply having household 4 bring its notebook into the office, where there was good connectivity between devices, allowed its data to be fully replicated. For household 10, we attached a one terabyte external drive to an active device that had data to be replicated. After doing this, less than 0.5% percent of files remained to be replicated.

Several households (1–5, 7 and 9) were able to achieve a replication factor greater than two for some of their files, enabling these files to survive multiple device failures. In Household 2, 80% of the user files were replicated 4 times or more.

6.5.2 Availability results

A secondary goal of PodBase is to place replicas on devices where they are likely to be useful. Specifically, our mp3 plug-in causes music files to be preferentially placed on devices that are capable of playing music.

In analyzing the trace, we found that one household had no mp3s and three households had already replicated all of their music files on the relevant devices. Thus, PodBase did not have an opportunity to improve availability. However, it did provide a significant gain in availability for several other households. Household 3 had its entire music library of 415 music files made available on all three of its devices. Households 7 and 8 had 851 and 1318 music files made available by PodBase, respectively. Household 9 had 1500 music files from a music library, which was otherwise loosely synchronized between its devices, made available on two additional devices. An additional two households originally had a significant number of mp3 files on their laptops but not on their desktops. PodBase replicated these files onto the desktops, and the mp3 plug-in described in section 4.4 had externalized the music files. This happened during an earlier run of PodBase, therefore it did not show up in our trace. The users gained access to 426 songs and 2611 songs, respectively, on their desktop computers. (The songs were previously stored only on their notebooks.)

As described in Section 5.2, the replication first optimizes for durability, then cost (time to complete), and finally availability. A concern might be that this choice limits the availability the system can provide. We looked at the impact of this optimization process on household 9, for which the final replication plan had not achieved full replication for availability. In this household the final replication plan yields 95% of the optimal availability. The remaining 5% were not achieved because the replication had not yet finished at the end of the trace, and not because of a limitation in the algorithm.

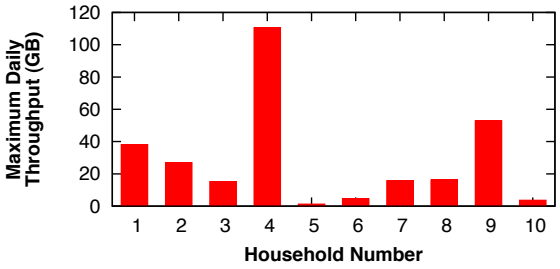


Figure 4: Peak daily throughput for each household

because a bug was discovered during the user study that had influenced the final state of this household

6.5.3 Replication latency and throughput

We next look at the maximal replication throughput in each of the households. Since all households had many files to replicate at the beginning of the trace collection, the rate at which data was replicated early in the trace is a lower bound for the total replication throughput of a pool. This value in turn provides a lower bound for the rate of new or modified data that a household could generate, such that PodBase would still be able to keep up with replicating.

Figure 4 shows that the peak throughput ranges from 1.4 to 110 GB per day. This result shows that PodBase can keep up with a high to very high rate of data generation, using only existing pair-wise connectivity.

We now examine the replication latency, i.e., the elapsed time until a new or modified file becomes replicated. If a file is not yet replicated at the end of the trace, we include it in the CDF as having an infinite latency. We first examine those households with relatively short latencies. Figure 5(a) shows a CDF of how long it took to replicate a file. For households 2 and 4, over 50% of files were replicated within approximately one day. Households 1 and 7 took longer because there were extended periods with no connectivity. Household 9 replicated gradually over the course of the trace, as connectivity allowed. Second, we show the latency of the households that took significantly longer to replicate their files in Figure 5(b). In these households, device connectivity is the dominant factor in the replication latency. When there is connectivity, there are sharps jumps as files get replicated, followed by periods of disconnection, where no replication happens.

We note that our measured replication latencies are conservative, because in most households, PodBase was busy replicating the user files found initially on the devices during a large part of the trace collection. In steady state, PodBase would have to replicate only newly created or modified files, reducing the latencies considerably. Nevertheless, PodBase was able to replicate data in a timely fashion, subject to available storage and device connectivity.

6.5.4 Rate of new or modified data

Next, we look at the rate of new or modified data that is being generated. Each of the households in the user study had on average 528,187 files taking up 332GB. After the initial crawl, an average of 21GB per day was generated by the addition of new and modifications of existing files. These numbers are skewed by a household that stored the disk image of an active virtual machine in the file system; without this household, the value was 381MB per day. (Of course, PodBase could be optimized to handle this case more efficiently.)

Our normal households generate new or modified data at a minimal/average/maximal rate of 4.5/36.1/316 Kb/s, while the "heavy" household generates 2.3 Mb/s. Let us consider how well a backup system based on cloud storage alone would perform in our households. At an assumed broadband upload bandwidth of 1 Mb/s, transferring the initial data to the cloud while keeping up with updates would require between 3.7 and 121.6 (median 31.82) days for the normal households. For the heavy household, cloud storage would be infeasible, because the rate of new data exceeds the network bandwidth.

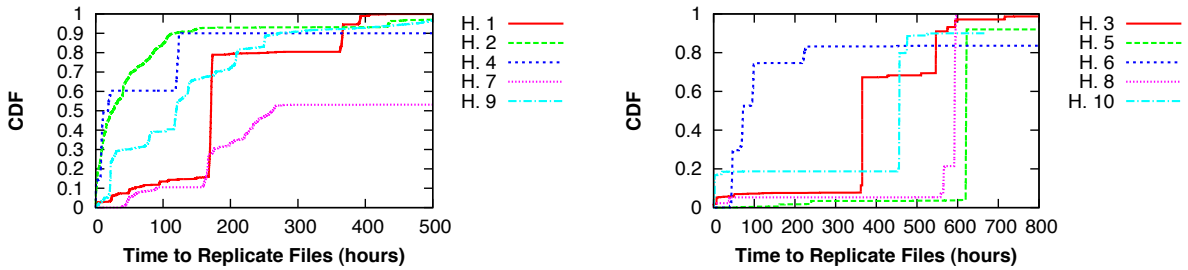
These results show that for timely replication of data, PodBase's use of peer connections and local storage devices is important. For the normal households, a broadband connection would suffice to replicate new data, but the heavy household would require a faster Internet connection. Even for the normal households, relying solely on a broadband connection to the cloud would require a long period of full network utilization to replicate the initial data, and increase the replication latency in steady state (and therefore the window of vulnerability for new and modified files that have not yet been replicated).

6.6 Discussion

PodBase has been developed by the first author over a period of two years, with three user deployments at different stages. Significant engineering effort was required to make sure our users (most of whom where not affiliated with the project) and their families felt comfortable running it on their personal devices. Users demanded not to have to notice the presence of the system in their daily activity or be surprised by it actions, yet expected the system to do "the right thing" without requiring their attention. Moreover, different households used their devices in very different ways, some of which we could not have imagined (see the discussion of results for different households in Section 6.5). This forced us to emphasize non-intrusiveness (not interfering with user's activities), autonomy (making reasonable choices without user's input) and adaptivity to unexpected scenarios far more than efficiency. Apart from the quantitative results reported in this section, the most important indicator of the project's success may be the fact that ten households (which included members who had little interest and patience for our project) agreed to use the system for the duration of the study and beyond.

7 Conclusion

PodBase transparently manages the data stored on personal devices for durability and availability. The system takes advantage of existing free storage space and incidental connectivity among devices. Thus, it reduces the



(a) Replication latency for households 1,2,4,7,9

(b) Replication latency for households 3,5,6,8,10.

Figure 5: Replication latency for all households

need for dedicated backup storage or an external storage provider and avoids the bottleneck of a home broadband uplink. PodBase relies on optimization techniques to achieve highly adaptive replication. The system is fully decentralized and does not depend on the health of any one device. Experimental results from a user deployment in ten real households indicate that the system is effective in replicating data without any user attention, and in many cases without requiring additional storage.

Acknowledgements

We are hugely indebted to the volunteers who participated in our user study, who patiently stuck with us through multiple deployments. Also, we wish to thank the anonymous reviewers of this and previous versions of this paper, as well as our shepherd Ed Nightingale, for their helpful comments and suggestions.

References

[1] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc. NSDI '06*, 2006.

[2] B. Cho and I. Gupta. New algorithms for planning bulk transfer via internet and shipping networks. In *Proc. ICDCS '10*, ICDCS '10, 2010.

[3] COIN-OR Linear Programming Solver. `http://projects.coin-or.org/Clp`.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. *SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[5] Dropbox. `https://www.dropbox.com/`.

[6] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proc. SIGCOMM '03*, Aug 2003.

[7] A. D. Fekete and K. Ramamritham. Replication. volume 5959 of *Lecture Notes in Computer Science*, in chapter Consistency Models for Replicated Data, pages 1–17. Springer Berlin / Heidelberg, 2010.

[8] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent personal names for globally connected mobile devices. In *Proc. OSDI '06*, Nov 2006.

[9] Groove. `http://office.microsoft.com/groove`.

[10] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *Proc. PerCom '06*, Mar 2006.

[11] K. Keeton, T. Kelly, A. Merchant, C. Santos, J. Wiener, X. Zhu, and D. Beyer. Don't settle for less than the best: Use optimization to make decisions. In *Proc. HotOS '07*, May 2007.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[13] Live mesh. `http://www.mesh.com`.

[14] Many PC Users don't backup valuable data. `http://money.cnn.com/2006/06/07/technology/data_loss/index.htm`.

[15] Mozy. `https://www.mozy.com/`.

[16] J. P. Munson and P. Dewan. A flexible object merging framework. In *Proc. CSCW '94*, pages 231–242, New York, NY, USA, 1994. ACM.

[17] PC Pitstop Research. `http://pcpitstop.com/research/storagesurvey.asp`.

[18] D. Peek and J. Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proc. OSDI '06*, Nov 2006.

[19] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann. Replication in Ficus distributed file systems. In *Proc. WMRD*, pages 20–25, Nov 1990.

[20] A. Post, P. Kuznetsov, and P. Druschel. PodBase: Transparent storage management for personal devices. In *Proc. IPTPS '08*, Feb 2008.

[21] A. Post, J. Navarro, P. Kuznetsov, and P. Druschel. Automomous storage management for personal devices with Podbase. Technical Report 001, MPI-SWS, 2011. `http://www.mpi-sws.org/tr/2011-001.pdf`.

[22] N. Preguiça, C. Baquero, J. L. Martins, M. Shapiro, P. S. Almeida, H. Domingos, V. Fonte, and S. Duarte. Few: File management for portable devices. In *Proc. IWSSPS 2005*, March 2005.

[23] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: a platform for content-based partial replication. In *Proc. NSDI'09*, 2009.

[24] D. Ratner, P. Reiher, and G. J. Popeky. Roam: A scalable replication system for mobility. *Mobile Networks and Applications*, 9(5):537–544, 2004.

[25] M. Rodrig and A. LaMarca. Oasis: an architecture for simplified data management and disconnected operation. *Personal and Ubiquitous Computing*, 9(2):108–121, 2005.

[26] Y. Saito and M. Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, Mar. 2005.

[27] B. Salmon, S. W. Schlosser, L. B. Mummert, and G. R. Ganger. Putting home storage management into perspective. Technical Report CMU-PDL-06-110, Parallel Data Laboratory, Carnegie Mellon University, 2006.

[28] Skydrive. `http://skydrive.live.com/`.

[29] S. Sobti, N. Garg, F. Zheng, J. Lai, Y. Shao, C. Zhang, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: A distributed mobile storage system. In *Proc. FAST '04*, March 2004.

[30] J. Strauss, C. Lesniewski-Laas, J. M. Paluska, B. Ford, R. Morris, and F. Kaashoek. Device transparency: a new model for mobile storage. *SIGOPS Oper. Syst. Rev.*, 44(1):5–9, 2010.

[31] SugarSync. `http://www.sugarsync.com/`.

[32] E. Swierk, E. Kıcıman, N. C. Williams, T. Fukushima, H. Yoshida, V. Laviano, and M. Baker. The Roma Personal Metadata Service. In *Proc. WMCSA 2000*.

[33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP'95*, December 1995.

[34] Time Machine. `http://www.apple.com/macosx/features/timemachine.html`.

[35] D. N. Tran, F. Chiang, and J. Li. Friendstore: Cooperative online backup using trusted nodes. In *SocialNet'08: First International Workshop on Social Network Systems*, Glasgow, Scotland, 2008.

[36] Unison File Synchronization. `http://www.cis.upenn.edu/~bcpierce/unison/`.

[37] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The Personal Server: Changing the way we think about ubiquitous computing. In *Proc. of Ubicomp'02*, 2002.

[38] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. In *Proc. HotNets-II*, 2003.

[39] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, September 1991.

[40] Windows home server. `http://www.microsoft.com/windows/products/winfamily/windowshomeserver/default.mspx`.

[41] Windows live sync. `http://sync.live.com/`.

[42] Winfs team blog. `http://blogs.msdn.com/winfs/`.

[43] Q. Yin, J. Cappos, A. Baumann, and T. Roscoe. Dependable self-hosting distributed systems using constraints. In *Proc. HotDep '08*, Dec 2008.