USENIX

Proceedings of the 2010 USENIX Annual Technical Conference

Boston, MA, USA *June 23–25, 2010*

conference

proceedings

Sponsored by

USENIX

2010 USENIX Annual Technical Conference (USENIX ATC '10)

Boston, MA, USA

June 23–25, 2010

© 2010 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 978-931971-75-1

USENIX Association

Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC '10)

June 23–25, 2010
Boston, MA, USA

Conference Organizers

Program Co-Chairs

Paul Barham, *Microsoft Research Cambridge*
Timothy Roscoe, *ETH Zürich*

Program Committee

LKaterina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*
Andrew Baumann, *ETH Zürich*
Frank Bellosa, *University of Karlsruhe*
Ranjita Bhagwan, *Microsoft Research India*
Mic Bowman, *Intel Digital Enterprise Group*
Kevin Elphinstone, *University of New South Wales and NICTA*
Kevin Fall, *Intel Labs Berkeley*
Garth Gibson, *Carnegie Mellon University and Panasas*
Garth Goodson, *NetApp, Inc.*
Steven Hand, *University of Cambridge and Citrix Systems*
Hermann Härtig, *Technische Universität Dresden*
Galen Hunt, *Microsoft Research Redmond*
Michael Isard, *Microsoft Research Silicon Valley*
Jaeyeon Jung, *Intel Labs Seattle*
Orran Krieger, *VMware, Inc.*
Shan Lu, *University of Wisconsin—Madison*
Z. Morley Mao, *University of Michigan*
Dave Presotto, *Google, Inc.*
John Regehr, *University of Utah*
Prashant Shenoy, *University of Massachusetts Amherst*
Emin Gün Sirer, *Cornell University*
Michael Stumm, *University of Toronto*
Andrew Warfield, *University of British Columbia and Citrix Systems*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
Xiaolan (Catherine) Zhang, *IBM T.J. Watson Research Center*

The USENIX Association Staff

External Reviewers

Brendan Cully
Mike Dahlin
Austin Donnelly
Gernot Heiser
Dutch Meyer
Stuart Schechter
Michael Vrable

2010 USENIX Annual Technical Conference
June 23–25, 2010
Boston, MA, USA

Message from the Program Co-Chairs

Welcome to the 2010 USENIX Annual Technical Conference!

This year continues the USENIX Annual Tech tradition of papers and presentations reflecting some of the finest practical research work in computer systems today. The program committee has put together a highly selective program of 24 excellent papers—22 full papers and 2 short papers—from a total of 147 submissions. The topics cover a broad range, including many new and burgeoning areas of research such as energy reduction, information security, multicore systems, cloud computing, and solid-state mass storage. Furthermore, USENIX Annual Tech this year features an expanded selection of distinguished guest speakers, including Turing Award winner Ivan Sutherland, Bobby Johnson of Facebook, and Processing author Ben Fry.

We were privileged to be able to work with a great program committee of 27 members from a range of international industrial and academic research institutions. Every paper submitted to the conference received at least three reviews. Based on these, 94 papers were selected for a second round of reviewing in which each received at least one and, in most cases, two more full reviews. The majority of these reviews were written by the committee, who wrote an average of 22.5 reviews each for the conference, but we were also helped by the expertise of seven external reviewers. In total, 607 reviews were written. The committee then met in Redmond, Washington, on March 12, 2010, for an all-day discussion of the second-round papers which resulted in the final program. Each of the accepted papers was shepherded by a PC member.

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 at short notice. We are also grateful to Microsoft for hosting the PC meeting in Redmond and to Eddie Kohler for the HotCRP conference management software, which continues to make the process of preparing a program dramatically smoother than in the past. Finally, the USENIX staff—Ellie Young, Jane-Ellen Long, Devon Shaw, Casey Henderson, and many others—have worked tirelessly behind the scenes to make the conference a success.

We would also like to thank our industry sponsors for their support in making the 2010 USENIX Annual Technical Conference possible and enjoyable. In particular we thank our Silver Sponsors EMC^2, Facebook, Microsoft Research, and VMware, and our Bronze Sponsors Google and NetApp, as well as our Industry Partner and our Media Sponsors.

We hope you enjoy the program and the conference!

Paul Barham, ***Microsoft Research Cambridge***
Timothy Roscoe, ***ETH Zürich***

DEFCON: High-Performance Event Processing with Information Security

Matteo Migliavacca
Department of Computing
Imperial College London

Ioannis Papagiannis
Department of Computing
Imperial College London

David M. Eyers
Computer Laboratory
University of Cambridge

Brian Shand
CBCU, Eastern Cancer Registry
National Health Service UK

Jean Bacon
Computer Laboratory
University of Cambridge

Peter Pietzuch
Department of Computing
Imperial College London

smartflow@doc.ic.ac.uk

Abstract

In finance and healthcare, event processing systems handle sensitive data on behalf of many clients. Guaranteeing information security in such systems is challenging because of their strict performance requirements in terms of high event throughput and low processing latency.

We describe DEFCON, an event processing system that enforces constraints on event flows between event processing units. DEFCON uses a combination of static and runtime techniques for achieving light-weight isolation of event flows, while supporting efficient sharing of events. Our experimental evaluation in a financial data processing scenario shows that DEFCON can provide information security with significantly lower processing latency compared to a traditional approach.

1 Introduction

Applications in finance, healthcare, systems monitoring and pervasive sensing that handle personal or confidential data must provide both strong security guarantees and high performance. Such applications are often implemented as event processing systems, in which flows of event messages are transformed by processing units [37]. Preserving information security in event processing without sacrificing performance is an open problem.

For example, financial data processing systems must support high message throughput and low processing latency. Trading applications handle message volumes peaking in the tens of thousands of events per second during the closing periods on major stock exchanges, and this is expected to grow in the future [1]. Low processing latency is crucial for statistical arbitrage and high frequency trading; latencies above a few milliseconds risk losing the trading initiative to competitors [12].

At the same time, information security is a major concern in financial applications. Internal proprietary traders have to shield their buy/sell message flows and trading strategies from each other, and be shielded themselves from the client buy/sell flows within a bank. Information leakage about other buy/sell activities is extremely valuable to clients, as it may lead to financial gain, motivating them to look for leaks. Leakage of client data to other clients may damage a bank's reputation; leakage of such data to a bank's internal traders is illegal in most jurisdictions, violating rules regarding conflicts of interest [8]. The UK Financial Service Authority (FSA) repeatedly fines major banks for trading on their own behalf based on information obtained from clients [15].

Traditional approaches for isolating information flows have limitations when applied to high-performance event processing. Achieving isolation between client flows by allocating them to separate physical hosts is impractical due to the large number of clients that use a single event processing system. In addition, physical rack space in data centres close to exchanges, a prerequisite for low latency processing, is expensive and limited [23]. Isolation using OS-level processes or virtual machines incurs a performance penalty due to inter-process or inter-machine communication, when processing units must receive multiple client flows. This is a common requirement when matching buy/sell orders, performing legal auditing or carrying out fraud detection. The focus on performance means that current systems do not guarantee end-to-end information security, instead leaving it to applications to provide their own, ad hoc mechanisms.

We enforce information security in event processing using a uniform mechanism. The event processing system prevents incorrect message flows between processing units but permits desirable communication with low latency and high throughput. We describe DEFCON, an event processing system that supports *decentralised event flow control* (DEFC). The DEFC model applies information flow control principles [27] to high-performance event processing: parts of event messages are annotated with appropriate security labels. DEFCON tracks the "taint" caused by messages as they flow through processing units and prevents information leakage when units lack appropriate privileges by controlling the external visibility of labelled messages. It also avoids the inference of information through implicit information flows—the absence of a unit's messages after that unit becomes tainted would otherwise be observable by other units.

To enforce event flow control, DEFCON uses application-level virtualisation to separate processing units. DEFCON isolates processing units within the same address space using a modified Java language runtime. This lightweight approach allows efficient communication between isolation domains (or *isolates*). To separate isolates, we first statically determine potential storage channels in Java, white-listing safe ones. After that, we add run-time checks by weaving interceptors into potentially dangerous code paths. Our methodology is easily reproducible; it only took us a few days to add isolation to OpenJDK 6.

Our evaluation using a financial trading application demonstrates a secure means of aggregating clients' buy/sell orders on a single machine that enables them to trade at low latency. Our results show that this approach gives low processing latencies of 2 ms, at the cost of a 20% median decrease in message throughput. This is an acceptable trade-off, given that isolation using separate processes results in latencies that are almost four times higher, as shown in §6.

In summary, the main contributions of the paper are:

- a model for decentralised event flow control in event processing systems;
- Java isolation with low overhead for inter-isolate communication using static and runtime techniques;
- a prototype DEFCON implementation and its evaluation in a financial processing scenario.

The next section provides background information on event processing, security requirements and related work on information flow control. In §3, we describe our model for decentralised event flow control. Our approach for achieving lightweight isolation in the Java runtime is presented in §4. In §5, we give details of the DEFCON prototype system, followed by evaluation results in §6. The paper finishes with conclusions (§7).

2 Background

2.1 Event processing

Event processing performs analysis and transformation of flows of event messages, as found in financial, monitoring and pervasive applications [24]. Since events are caused by real-world phenomena, such as buy/sell orders submitted by financial traders, event processing must occur in near real-time to keep up with a continuous flow of events. Popular uses of event processing systems are in fraud detection, Internet betting exchanges [7] and, in the corporate setting, for enterprise application integration and business process management [5]. While we focus on centralised event processing in this paper, event processing also finds applicability in-the-large to integrate "systems of systems" by inter-connecting applications without tightly coupling them [26].

Event processing systems, such as Oracle CEP [38], Esper [14] and Progress Apama [2], use a message-driven programming paradigm. *Event messages* (or *events*) are exchanged between *processing units*. Processing units implement the "business logic" of an event processing application and may be contributed by clients or other third-parties. They are usually reactive in design—events are dispatched to processing units that may emit further events in response. There is no single data format for event messages, but they often have a fixed structure, such as key/value pairs.

Financial event processing. In modern stock trading, low processing latency is key to success. As financial traders use automated algorithmic trading, response time becomes a crucial factor for taking advantage of opportunities before the competition do [20]. To support algorithmic trading, stock exchanges provide appropriate interfaces and event flows. To achieve low latency, they charge for the service of having machines physically co-located in the same data centre as parts of the exchange [16].

It was recently suggested that reducing latency by 6 ms may cost a firm \$1.5 million [9]. The advantage that they get from reacting faster to the market than their competition may translate to increased earnings of \$0.01 per share, even for trades generated by other traders [12]. However, even with co-location within the same data centre rack, there is a minimum latency penalty due to inter-machine network communication.

Therefore having multiple traders acting for competing institutions share a single, co-located machine has several benefits. First, trading latency is reduced since client processing may be placed on the same physical machine as the order matching itself [34]. Second, the traders can share the financial burden of co-location within the exchange. Third, they can carry out *local brokering* by matching buy/sell orders among themselves—a practice known as a "dark pool"—thus avoiding the commission costs and trading exposure when the stock exchange is involved [44].

Hosting competing traders on the same machine has significant security implications. To avoid disclosing proprietary trading strategies, each trader's stock subscriptions and buy/sell order feeds must be kept isolated. The co-location provider must respect clients' privacy; bugs must never result in information leakage.

2.2 Security in event processing

Today's event processing systems face challenging security requirements as they are complex, process sensitive data and support the integration of third-party code. This increases the likelihood of software defects exposing information. Information leaks have serious consequences because of the sensitive nature of data in domains such as finance or healthcare. As in the stock-trading platform example, the organisation providing the event processing service is frequently not the owner of the processed data. Processing code may also be contributed by multiple parties, for example, when trading strategies are implemented by the clients of a trading platform.

Event processing systems should operate according to data security policies that specify system-wide, end-to-end *confidentiality* and *integrity* guarantees. For example, traders on a trading platform require their trading strategies not to be exposed to other traders (confidentiality). The input data to a trading strategy should only be stock tick events provided by the stock exchange (integrity). This cannot be satisfied by simple access control schemes, such as access control lists or capabilities, because they alone cannot give end-to-end guarantees: any processing unit able to access traders' orders may cause a leak to other traders due to bugs or malicious behaviour. Anecdotal evidence from the (rather secretive) financial industry, and existing open source projects [35], suggest that current proprietary trading systems indeed lack mechanisms to enforce end-to-end information security. Instead, they rely on the correctness (and compliant behaviour) of processing units.

Threat model. We aim to improve information security in event processing by addressing the threat that information in events may be perceived or influenced by unauthorised parties. Our threat model is that processing units may contain unintentional bugs or perform intentional information leakage. We do not target systems that run arbitrary code of unknown provenance: event processing systems are important assets of organisations and are thus carefully guarded. Only accountable parties are granted access to them. As a consequence, we are not concerned about denial-of-service attacks from timing-related attacks or misuse of resources—we leave protection against them for future work. However, we do want protection from parties that may otherwise be tempted not to play by the rules, e.g. by trying to acquire information that they should not access or leak information that they agreed to keep private. We assume that the operating system, the language runtime and our event processing platform can be trusted.

2.3 Information flow control

We found that information flow control, which provides fine-grained control over the sharing of data in a system, is a natural way to realise the aforementioned kind of security that event processing systems require.

Information flow control is a form of mandatory access control: a principal that is granted access to information stored in an object cannot make this information available to other principals, for example, by storing the information in an unprotected object (no-write-down or *-property) [6]. It was initially proposed in the context of military multi-level security [11]: principals and objects are assigned security *labels* denoting levels, and access decisions are governed by a "can-flow-to" partial order. For example, a principal operating at level "secret" can read a "confidential" object but cannot read a "top-secret" or write to a "confidential" object. Through this model, a system can enforce confinement of "secret" information to principals with "secret" (or higher) clearance.

Equivalently, IFC-protected objects may be thought of as having a *contaminating* or *tainting* effect on the principals that process them—a principal that reads a "secret" document must be contaminated with the "secret" label, and will contaminate all objects it subsequently modifies.

Compartments created by labels are fairly coarse-grained and declassification of information is performed outside of the model by a highly-trusted component. Myers and Liskov [27] introduce *decentralised information flow control* (DIFC) that permits applications to partition their rights by creating fresh labels and controlling declassification privileges for them. Jif [28] applies the DIFC model to variables in Java. Labels are assigned and checked statically by a compiler that infers label information for expressions and rejects invalid programs. In contrast, event-processing applications require fresh labels at runtime, for example, when new clients join the system. Trishul [29] and Laminar [32] use dynamic label checks at the JVM level. However, tracking flows between variables at runtime considerably reduces performance.

Myers and Liskov's model also resulted in a new breed of DIFC-compliant operating systems that use labels at the granularity of OS processes [13, 43, 22]. Asbestos [13] enables processes to protect data and enforces flow constraints at runtime. Processes' labels are dynamic, which requires extra care to avoid implicit information leakage, and Asbestos suffers from covert storage channels. HiStar [43] is a complete OS redesign based on DIFC to avoid covert channels. Flume [22] brings DIFC to Linux by intercepting system calls and augmenting them with labels. All of the above projects isolate processes in separate address spaces and provide IPC abstractions for communication. For event processing, this would require dispatching events to processing units by copying them between isolates, resulting in lower performance (cf. §6).

The approach closest to ours is Resin [41], which discovers security vulnerabilities in applications by modifying the language runtime to attach data flow policies to data. These policies are checked when data flows cross guarded boundaries, such as method invocations. Resin only tracks the policy when data is explicitly copied or altered, making it unsuitable to discover deliberate, implicit leakage of information, as it may be found in financial applications.

3 DEFCon Design

This section describes the design of our event processing system in terms of our approach for controlling the flow of events. We believe that it is natural to apply information flow constraints at the granularity of events because they constitute explicit data flow in the system. This is in contrast to applying constraints with operating system objects or through programming language syntax extensions, as seen in related research [13, 43, 22, 27].

3.1 DEFC model

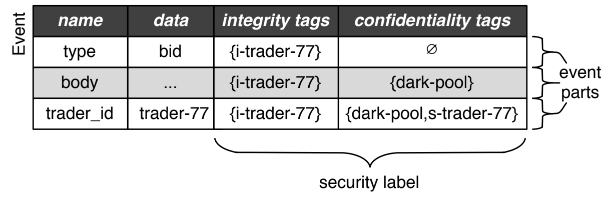
We first describe our model of *decentralised event flow control* (DEFC). The DEFC model uses information flow control to constrain the flow of events in an event processing system. In this paper, we focus on aspects of the model related to operation within a single machine as opposed to a distributed system.

The DEFC model has a number of novel features, which are specifically aimed at event processing: (1) multiple labels are associated with parts of event messages for fine-grained information security (§3.1.2); (2) privileges are separated from privilege delegation privileges—this lets event flows be constrained to pass through particular processing units (§3.1.3); (3) privileges can be dynamically propagated using privilege-carrying events, thus avoiding implicit, covert channels (§3.1.5); and (4) events can be partially processed by units without tainting all event parts (§3.1.6).

3.1.1 Security labels

Event flow is monitored and enforced through the use of *security labels* (or *labels*), which are similar to labels in Flume [22]. Labels are the smallest structure on which event flow checking operates, and protect confidentiality and integrity of events. For example, labels can act to enforce isolation between traders in a financial application, or to ensure that particularly sensitive aspects of patient healthcare data are not leaked to all users.

As illustrated in Figure 1, security labels are pairs, (S, I), consisting of a *confidentiality component* S and an *integrity component* I. S and I are each sets of *tags*. Each tag is used to represent an individual, indivisible concern either about the privacy, placed in S, or the integrity, placed in I, of data. Tags are opaque values,



name	data	integrity tags	confidentiality tags
type	bid	{i-trader-77}	∅
body	...	{i-trader-77}	{dark-pool}
trader_id	trader-77	{i-trader-77}	{dark-pool,s-trader-77}

Figure 1: An event message with multiple named parts, each containing data protected by integrity and confidentiality tags.

implemented as unique, random bit-strings. We refer to them using a symbolic name, such as i-trader-77 (an integrity tag in this case).

Tags in confidentiality components are "sticky": once a tag has been inserted into a label component, data protected by that label cannot flow to processing units without that tag, unless privilege over the tag is exercised. In contrast, tags in integrity components are "fragile": they are destroyed when information with such tags is mixed with information not containing the tag, again unless a privilege is exercised.

For example, if a processing unit in a trading application receives data from two other units with confidentiality components {s-trading, s-client-2402} and {s-trading, s-trader-77} respectively, then any resulting data will include all of the tags {s-trading, s-client-2402, s-trader-77}. This reflects the sensitivity with respect to both sources of the data. Similarly, if data from a stock ticker with an integrity component {i-stockticker} is combined with client data with integrity {i-trader-77}, the produced data will have integrity {}. This shows that the data cannot be identified as originating directly from the stock ticker any more.

Labels form a lattice: for the confidentiality component (S), information labelled S_a can flow to places holding component S_b if and only if $S_a \subseteq S_b$; here $\subseteq$ is the "can flow to" ordering relation [42]. For integrity labels (I), "can flow to" order is the superset relation $\supseteq$. Thus we define the "can flow to" relationship $L_a \prec L_b$ for labels as:

$$L_a \prec L_b \quad \text{iff} \quad S_a \subseteq S_b \text{ and } I_a \supseteq I_b$$
$$\text{where } L_a = (S_a, I_a) \text{ and } L_b = (S_b, I_b)$$

3.1.2 Anatomy of events

A key aspect of our model is the use of information flow control at the granularity of events. An event consists of a number of event *parts*. Each part has a *name*, associated *data* and a *security label*. Using parts within an event allows it to be processed by the system as a single, connected entity, but yet to carry data items within its parts that have different security labels. Dispatching a single event with secured parts supports the principle of least privilege—processing units only obtain access to parts of the event that they require.

Figure 1 shows a bid event in a financial trading application with three parts. The event is tagged with the

trader's integrity tag. The information contained in the bid has different sensitivity levels: the type part of the event is public, while the body part is confined to match within the dark pool by the dark-pool tag. The identity part of the trader is further protected by a trader-private confidentiality tag.

Access to event parts is controlled by the system that implements DEFC. When units want to retrieve or modify event parts, or to create new events, they must use an API such as the one described in §5.

3.1.3 Constraining tags and labels

Each processing unit can store state—its data can persist between event deliveries. Rather than associate labels with each piece of state in that unit, a single label (S_u, I_u) is maintained with the overall confidentiality and integrity of the unit's state. (We also refer to this as the unit's *contamination level*.) This avoids the need for specific programming language support for information flow control, as most enforcement can be done at the API level.

The ability of a unit to add or remove a tag to/from its label is a *privilege*. A unit u's run-time privileges are represented using two sets: O_u^+ and O_u^-. If a tag appears in O_u^+, then u can add it to S_u or I_u. Likewise, u can remove any tag in O_u^- from any of its components.

If unit u adds tag $t \in O_u^+$ to S_u, then t is used as a confidentiality tag, moving u to a higher level of secrecy. This lets u "read down" no less (and probably more) data than before. If t is used as an integrity tag, then adding it to I_u would be exercising an *endorsement privilege*. Conversely, removing a confidentiality tag $t \in O_u^-$ from S_u involves unit u exercising a *declassification privilege*, while removing an integrity tag t from I_u is a transition to operation at lower integrity.

For dynamic privilege management, privileges over tag privileges themselves are represented in two further sets per unit: $O_u^{-\text{auth}}$ and $O_u^{+\text{auth}}$. We define their semantics with a short-hand notation: t_u^+ means that $t \in O_u^+$; t_u^- means $t \in O_u^-$; $t_u^{+\text{auth}}$ means $t \in O_u^{+\text{auth}}$; $t_u^{-\text{auth}}$ means $t \in O_u^{-\text{auth}}$ for tag t and unit u. We will omit the u subscript when the context is clear.

$t_u^{-\text{auth}}$ lets u *delegate* the corresponding privilege over tag t to a target unit v. After delegation, t_v^- holds. Likewise for $t_u^{+\text{auth}}$. If $t_u^{-\text{auth}}$, u can also delegate to v the ability to delegate privilege, yielding $t_v^{-\text{auth}}$ (likewise for $t_u^{+\text{auth}}$). Delegation is done by passing privilege-carrying events between units (cf. §3.1.5), ensuring that the DEFC model is enforced without creating a covert channel.

The separation of O_u^+ and $O_u^{+\text{auth}}$, in contrast to Asbestos/HiStar or Flume, allows our model to enforce specific processing topologies. For example, a Broker unit can send data to the Stock Exchange unit only through a Regulator unit, by preventing the Regulator from delegating to the Broker the right to communicate with the Stock Exchange directly.

Units can request that tags be created for them at runtime by the system. Although opaque to the units, tags and tag privilege delegations are transmittable objects. When a tag t is successfully created for a unit u, then $t_u^{-\text{auth}}$ and $t_u^{+\text{auth}}$. In many cases, u will apply these privileges to itself to obtain t_u^- or t_u^+.

A unit can have both t_u^- and t_u^+; then u has complete privilege over t. Note that the privilege alone does not let u transfer its privileges to other units.

3.1.4 Input/Output labels

Processing units need a convenient way to express their intention to use privileges when receiving or sending events. A unit u applies privileges by controlling an *input* label $(S_u^{\text{in}}, I_u^{\text{in}})$, which is equivalent to its contamination level (S_u, I_u), and an *output label* $(S_u^{\text{out}}, I_u^{\text{out}})$. Changes to these labels cause the system automatically to exercise privileges on behalf of the unit when it receives or sends events, in order to reach a desired level. Input/output labels increase convenience for unit programmers: they avoid repeated API calls to add and remove tags from labels when outputting events, or to change a unit's contamination label temporarily in order to be able to receive a given event.

For example, a Broker unit can add an integrity tag i to I_u^{out} but not to I_u^{in}. This enables it to vouch for the integrity of the stock trades that it publishes without having to add tag i explicitly each time. Similarly, adding tag t temporarily to S_u^{in} but not to S_u^{out} allows a Broker to receive and declassify t-protected orders without changing the code that handles individual events. In both cases, the use of privileges is only required when changing the input and output labels and not every time when handling an event.

Note that systems that allow for implicit contamination risk leaking information. For example, one could posit a model in which a unit's input and output labels rose automatically if that unit read an event part that included tags that were not within the unit's labels. The problem with this is that if unit u observes that it can no longer communicate with unit v that has been implicitly contaminated, then information has leaked to u. Therefore we require explicit requests for all changes to the input/output labels.

3.1.5 Dynamic privilege propagation

We use *privilege-carrying events* as an in-band mechanism to delegate privileges between processing units. A request to read a privilege-carrying part will bestow privileges on the requesting unit—but only if the unit already has a sufficient input label to read the data in that part.

An example of this is a Regulator unit trying to learn the identity of a trader mentioned in a trade event. The trader's identity is protected against disclosure by a unique tag t, but t^+ and t^- are included in another part visible to the Regulator unit only. This means that the Regulator can read this part, thus gaining t^+ and t^-, and then use these privileges to learn the trader's identity.

Although the bestowing of privileges is implicit, the privileges relate to a particular tag t, and the receiving unit cannot invoke the privileges without a reference to tag t itself. This reference is carried in the data part of an event: units, by design, will know in advance when to expect tags to be transferred to them, and when accessing a part will result in a privilege delegation. In the previous example, the tag t itself has to be in the data part that the Regulator accesses.

3.1.6 Partial event processing

Event processing frequently involves units transforming events along a *main dataflow path*, augmenting events as they flow through the system. To allow units to update only some parts of an event, we distinguish event processing on the main path from events generated by units themselves. In the former case, a unit that adds a part does not cause the labels of all parts of that event to change to the unit's output label. In the latter case, all parts' labels match the unit's output label.

For example, partial event processing enables a Broker unit to operate on orders without knowing the identity of the originating trader. The Broker can have access to some parts, such as the bid/ask price, and subsequently add new parts, such as a reason why an order was rejected, without being aware of or affecting a protected part with the trader identity.

When an event is dispatched to a unit, the unit may read and/or modify some parts but not others. The unit must then invoke a `release` API call, after which the event dispatcher may deliver the event to other units. Unaltered event parts do not need to have their labels changed. A released event must not cause additional deliveries to units with lower input labels. When multiple units make conflicting modifications to a part, the resulting event will have to contain both versions of the affected part.

3.2 DEFCON architecture

Our DEFCON architecture, that implements the DEFC model, is illustrated in Figure 2. The DEFCON system provides a runtime environment for a set of event processing units that implement the business logic of an event processing application. Units interact with the DEFCON system through API calls. As shown in the figure, the DEFCON system carries out the following tasks:

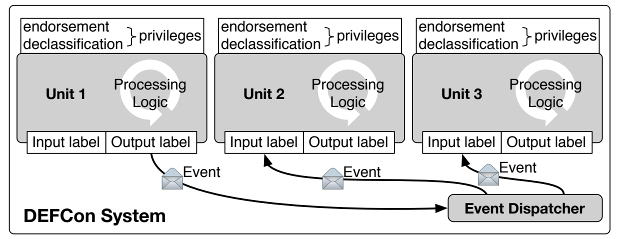


Figure 2: Overview of the DEFCON architecture.

Label/tag management. DEFCON maintains the set of defined tags in the *tag store*. It also keeps track of the input and output labels and privileges for each unit. The tags that make up labels are opaque to units. Units access tags by reference but cannot modify them directly.

Inter-unit communication. DEFCON provides units with a publish/subscribe API to send/receive events. To receive call-backs that provide event references, units register their interests by making subscriptions. An *event dispatcher* sends events to units that have expressed interest previously. This decoupled communication means that the fact that a publish call has succeeded does not convey any information that might violate DEFC (e.g. which units were actually notified).

Unit life-cycle management. DEFCON instantiates and terminates event processing units. Having DEFCON manage units allows it to apply restrictions to the operations that units can do, as described in the next section.

To enforce event flow control, DEFCON must prevent units from communicating directly except through the event dispatcher that can check DEFC constraints. Otherwise a unit with clearance to receive confidential events could avoid the confinement imposed by its label by using a communication channel that is not protected by labels. Therefore each unit must execute within its own *isolate* that prevents it from interacting with other units or components outside of the DEFCON system.

4 Practical, Light-weight Java Isolation

As described in §2.1, a requirement for DEFCON is to prevent unauthorised processing units from communicating with each other, while supporting low latency, high throughput event communication between permitted units. Making units separate OS-level processes achieves isolation but comes at the cost of increased communication latency due to inter-process communication, serialisation of potentially complex event message data and context switching overhead. In §6, we show that this results in higher processing latencies. Therefore, we isolate units executing within the same OS process through the introduction of new mechanisms within the programming language runtime.

We chose Java for our implementation because it is a mature, strongly-typed language that is representative of the languages used to build industrial-strength event processing applications. Processing units are implemented as Java classes, which means that they can communicate efficiently using a shared address space.

We assume that we have access to the Java bytecode of processing units and that they are implemented using the DEFCON API (cf. §5). As a consequence, we can prevent them from using any JDK libraries (e.g. for I/O calls) or Java features (e.g. reflection) that are not strictly necessary for event processing. However, units may still contain bugs that cause them to expose confidential events to other units during regular processing, or they may explicitly try to use events with confidential data as part of their own processing to gain an illicit advantage.

Enforcing isolation between Java objects is not a trivial task because Java was not designed with this need in mind. Even if two Java objects never explicitly shared an object reference, they can exploit a wide range of *covert channels* to exchange information and violate isolation. Covert channels can be classified into storage and timing channels. Storage channels involve objects using unprotected, shared state to exchange data. Therefore we must close storage channels in Java. Since timing channels, which are caused by the modulation of system resources, such as CPU utilisation, are harder to exploit in practice, we ignore them in this work.

There is a large number of existing storage channels in Java, which can be exploited in three fundamental ways: (1) There are about 4,000 *static fields* in the Java Development Kit (JDK) libraries (in OpenJDK 6). For example, a static integer `Thread.threadSeqNum` identifies threads, which can be altered to act as a channel between two classes; (2) Java contains more than 2,000 *native* methods, which may expose global state of the Java virtual machine (JVM) itself. Native methods of standard classes such as `String` and `Object` retrieve data from global, internal data structures of the JVM; and (3) Java has *synchronisation* primitives that enable classes to exchange one bit of information at a time.

Several proposals have been made for achieving isolation in Java. As we explain below, they do not satisfy both of our two main requirements:

Low manual effort. It should be easy to add isolation support to any production JVM, with a minimal number of manual code changes. Many projects have been discontinued due in part to the difficulty of keeping them synchronised with JDK updates;

Efficient inter-isolate communication. The communication mechanism between isolated processing units should allow message passing with low latency and high throughout.

4.1 Existing approaches

Isolation of shared state. Existing approaches to achieving Java isolation involve a great deal of manual work. Modifying production JDKs is a daunting task, while, in comparison, the overall performance of research JDKs is lacking. Certifying a JVM to be free of storage channels would require an exhaustive inspection.

J-Kernel [19] and Joe-E [25] prevent access to global state in an ad hoc way: they restrict user code from defining new classes that contain mutable static fields. For the JDK libraries, they prevent access to classes or methods that are found to expose global state. They achieve this by providing custom proxies to `System`, `File` and other classes.

KaffeOS [4] reports to have manually assessed all of the JDK classes with static fields. Classes were rewritten to remove static fields, re-engineered to be aware of isolates or "reloaded". Reloading unsafe classes in the JVM results in per-isolate instances of static fields. However, this reloading mechanism cannot be applied to classes that are transitively referenced by a shared class, such as `Object`, requiring the manual assessment of a large number of classes.

Sun's MVM [10] and I-JVM [17] avoid manual examination of static fields by transparently replicating all of them per isolate. The JVM is modified to keep replicated copies of static fields per isolate. It also tracks which isolate is currently executing, making corresponding replicas visible to that isolate. MVM is the only project that reports to have attempted a complete assessment of the native methods that can expose global state. The cost of repeating this process for each new JVM release is considerable and, since MVM was completed only on Solaris/SPARC and is no longer maintained, reproducing it without detailed knowledge of JVM internals is hard.

Inter-isolate communication. MVM (similar to `.NET` AppDomains [33]) uses a separate heap space per isolate, which requires serialisation of objects exchanged between isolates. Incommunicado [30] improves MVM's inter-isolate communication by using deep-copying in place of serialisation. These approaches limit the performance of event processing applications because they require message passing to copy data. As we show in §6, this nullifies many of the performance advantages of sharing an address space between isolates.

Efficient inter-isolate communication is supported by KaffeOS and I-JVM, which allow objects to be shared between isolates. However, this is not appropriate for enforcing event flow control because once two isolates have established a shared object, the system can no longer separate them when their labels change. J-Kernel and JX [18] provide an approach better suited to DEFC: they use indirection through a proxy for objects created in dif-

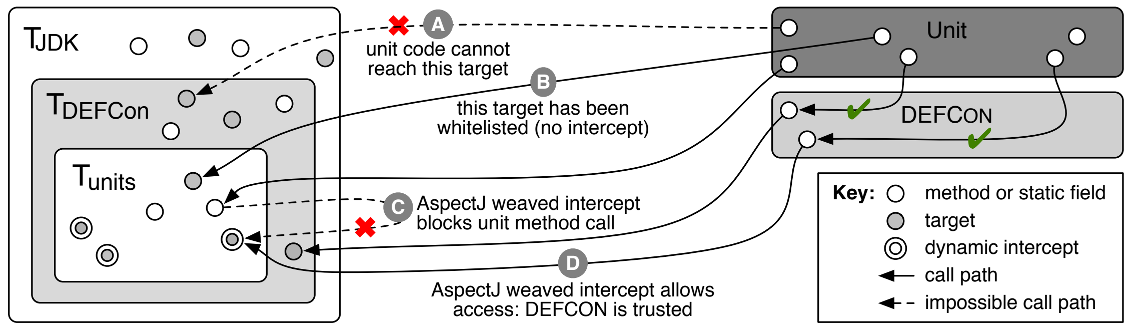

TJDK
TDEFCon
Tunits
A
unit code cannot reach this target
B
this target has been whitelisted (no intercept)
Unit
DEFCON
C
AspectJ weaved intercept blocks unit method call
D
AspectJ weaved intercept allows access: DEFCON is trusted
Key:
method or static field
target
dynamic intercept
call path
impossible call path

Figure 3: Illustrating our isolation enforcement between units using a combination of static white-listing and dynamic intercepts.

ferent isolates. However, their synchronous invocation model is at odds with decoupled event processing, which requires fast unidirectional communication.

4.2 Our isolation methodology

We describe a practical methodology for achieving Java isolation that provides fast, safe inter-isolate communication, while being easy to apply to new JDK versions. It does not require changes to the JVM or exhaustive code analysis.

We achieve efficient communication between isolates using message passing. Units do not have references to each other, only to objects controlled by DEFCON. For objects exchanged through events, we want to provide the semantics of passing objects by value, and exploit the single address space to avoid data copying. Our performance requirements preclude deep-copying of messages. Additionally, shared state is unacceptable because it violates isolation. Thus, we only allow units to exchange immutable objects, leaving it to units to perform copying only when needed.

We developed tools that help in the analysis of dangerous JDK *targets*: static fields, native methods and synchronisation primitives that could be used by units to communicate covertly. We were able to secure OpenJDK 6 in four days by manually inspecting only 52 targets (15 native methods, 27 static fields, and 10 synchronisation targets), without any modifications to the JVM.

As we illustrate in Figure 3, we divide potentially dangerous targets into three sets, T_{DEFCon}, T_{units} and T_{JDK}: a set of targets in the JDK only used by the DEFCON implementation (T_{DEFCon}), targets used by processing units (T_{units}), and targets used by neither (T_{JDK}). T_{units} was based on the event processing units that form the implementation of our trading platform described in §6.

Static dependency analysis. Targets not used at all (T_{JDK}), such as AWT/Swing classes, can be eliminated from the JDK without further impact. As a first step, we trim any classes that are not used by the DEFCON implementation or the event processing units of our financial scenario. This resulted in a subset of the JDK containing more than 2,000 used targets ($T_{\text{DEFCon}} \cup T_{\text{units}}$)—approximately 20% of the full JDK.

A significant proportion of these targets are only accessed by the DEFCON system (T_{DEFCon}) because they are not useful to units for processing events. Typically, (non-malicious) units use classes from the `java.lang` and `java.util` packages and have little reason to directly access classes from packages, such as `java.lang.reflect` or `java.security`. Thus we define a custom class loader that constrains the JDK classes that units can access to a *white-list*—e.g. precluding calls such as the one labelled 'A' in Figure 3.

However, restricting the set of classes alone does not prevent transitive access to dangerous targets. When the custom class loader permits the resolution of a white-listed JDK class, the loading of the class is delegated to the JVM bootstrap class loader. If the class contains references to other JDK classes, they are directly resolved by the JVM bootstrap classloader and therefore cannot be controlled.

Reachability analysis. In order to address the problem discussed above, a static analysis tool computes all targets that are transitively reachable from classes specified in the custom class loader white-list, i.e. T_{units} targets. This analysis enumerates possible method-to-method execution paths. The reachability analysis must cover code paths that involve dynamic method dispatch; a call to a given signature in the bytecode could execute code from any compatible subtype. Although the previous dependency analysis reduces the number of false positives in this phase, T_{units} still has 1,200 dangerous targets reachable from `java.lang`—approximately 320 native methods and 900 static fields.

Heuristic-based white-listing. Some of the targets in T_{units} can be declared safe using simple heuristics:

- We can white-list the 66 static fields and 20 native methods from the `Unsafe` class. This class provides direct access to JVM memory and is guarded by the Java Security Framework. Any access to it from user code would be a critical JVM bug.
- Some final static fields classified as immutable, such as strings or boxed primitive types, can be shared because they are constants.
- The use of some private static fields can be determined to be safe: vectors of constants and primitive fields that are not declared "final" but are only written once.

Another tool white-lists according to the above heuristics, reducing the number of dangerous targets to approximately 500 static fields and 300 native methods. Such cases are represented in Figure 3 by the call labelled 'B'.

Automatic runtime injection. To secure targets in T_{units} left after the preceding static analysis stage, we would have to duplicate unsafe static fields and manually assess native methods for covert communication channels, as done by other JVM isolation projects. In contrast to these projects, we wanted to avoid any JVM source code modification and to minimise the number of native JDK methods that needed to be checked.

For this reason, we employ aspect-oriented programming (AOP) [21]: by modifying JDK code in a programmatic way, we can duplicate static fields without changing the JVM and inject access checks to protect the execution of native methods. We employ the MAJOR/FERRARI framework [40] because it can manipulate JDK bytecode, as well as our own code, using the AspectJ language. We specify *pointcuts* to intercept all targets left after our static analysis, as follows:

Native methods: When access to a native target is as part of a call to the DEFCON API (described in §5), we can consider it safe by assuming the API is correctly designed (call 'D' in Figure 3). Otherwise we raise a security exception (call 'C').

Static fields: When a static field can be cloned without creating references that are shared with the original, we do an on-demand deep copy and create a per-unit reference. This occurs on a `get` access for most types, but can be deferred to the time of a `set` method for primitive or constant types. If field copying is not possible, we raise a security exception.

Manual white-listing. In this way, we automatically close JDK covert storage channels without changes to the JVM. However, before running the units in our financial scenario, we had to manually check 15 native methods and 27 static fields, which were intercepted and raised security exceptions. Below are a few examples of manually white-listed targets with a brief justification:

`java.lang.Object.hashCode:` This effect of this method is equivalent to reading a constant field.

`java.lang.Object.getClass:` Since `Class` objects are unique and constant, this method essentially retrieves a constant static field.

`java.lang.Double.longBitsToDouble:` This method does not access any JVM state.

`java.lang.System.security:` This target is safe because the reference to the security manager is protected from modification by units.

While the above methodology results in safe isolation, intercepting targets adds an overhead. We therefore profile the execution paths of units to identify frequently encountered targets that may be white-listed manually. During this profiling, we discovered 15 additional frequently-accessed targets (6 static fields and 9 native methods) that we were able to white-list.

4.3 Restricting synchronisation channels

As explained in §3.2, the DEFCON system must ensure that references held by one unit cannot escape to another unit. To avoid serialisation or deep-copying and to prevent the establishment of unrestricted shared state, units are limited to exchanging immutable objects whose references can be shared safely. However, every Java object, even if it is immutable, has a piece of modifiable information: its synchronisation lock. The lock is modified by `synchronized` blocks and by `wait` and `notify` calls.

This need to control synchronisation on shared objects also closes a further Java-specific channel due to the "interning" of strings. A string that has been interned is guaranteed to have a unique reference, common with all other strings of the same value in the JVM. This lets reference comparison (==) replace the more expensive `equals` method.

Previous proposals [10, 17] to avoid synchronisation on shared objects such as interned `String`s and `Class`es provide a copy per isolate. This would defeat the purpose of our message passing scheme that uses shared objects with the intent of avoiding copying them.

Automatic runtime injection. Instead we allow units to synchronise only on types that are guaranteed to never be shared with other units. This is indicated by the type in question implementing our `NeverShared` tagging interface. A type T can implement `NeverShared` as long as (a) the DEFCON system prevents instances of T being put into events, (b) no (white-listed) native method can return the same instance of T to two different units, and (c) no static field of type T is white-listed as being safe. Neither `Class` nor `String` objects satisfy these requirements and thus units cannot synchronise on them.

Units can instead make their own types for synchronisation that implement `NeverShared`. If a type is statically known to implement `NeverShared`, then synchronisation happens with no runtime overhead. Otherwise AOP will be used to inject a runtime type check: if this check fails and the attempt to synchronise comes from a unit, a security exception is raised.

DEFCON API call	Description
createEvent() $\rightarrow e$	Creates a new event e.
addPart(e, S, I, *name*, *data*)	Adds to event e a new part *name* containing *data* with label (S, I).
delPart(e, S, I, *name*)	Removes from event e part *name* with label (S, I)
readPart(e, *name*) $\rightarrow$ *(label, data)**	Returns the data in part *name* of event e. If there are multiple visible parts with the same *name*, all are returned. $S_p \subseteq S_u^{in}$ and $I_p \subseteq I_u^{in}$ must hold for every part returned to the unit.
attachPrivilegeToPart(e, *name*, S, I, t, p)	Attaches a privilege p over a tag t to part *name* with label (S,I) to create a privilege-carrying event for delegation (cf. §3.1.5). The call succeeds if the caller has t^{pauth}.
cloneEvent(e, S, I) $\rightarrow e'$	Creates a new instance e' of an existing event e. All the tags in the caller's output confidentiality label are attached to each part's label and only the caller's output integrity tags are maintained on each cloned part. This precludes DEFC violations based on observing the number of received events.
publish(e)	Publishes a new event e. Events without parts are dropped.
release(e)	Releases an event e (cf. §3.1.6).
subscribe(*filter*) $\rightarrow s$	Subscribes to events with a non-empty *filter*, creating a subscription s. The *filter* is an expression over the name and data of event parts. For an event to match, $S_p \subseteq S_u^{in}$ and $I_p \subseteq I_u^{in}$ must hold for each part in the filter at the time of matching.
subscribeManaged(*handler*, *filter*)$\rightarrow s$	Declares a *managed subscription* s that enables a unit to process multiple tags without contaminating its state permanently. DEFCON then creates and reuses separate unit instances with contaminations appropriate for the processing of incoming events. Units with managed subscriptions are similar to Asbestos' event processes [13].
getEvent() $\rightarrow (e, s)$	Blocks the caller until an incoming event e matches one of the unit's subscriptions s.
instantiateUnit(u', S, I, $O_{u'}^{p}$, $O_{u'}^{\mathrm{pauth}}$)	Instantiates a new unit u' at a given label (S, I), as long as it can delegate privileges to the new unit. The new unit inherits the caller's contamination.
changeOutLabel($\langle \mathrm{S} \vert \mathrm{I} \rangle$, $\langle \mathrm{add} \vert \mathrm{del} \rangle$, t)	Adds/removes tag t to/from a unit's output label $(S_u^{\mathrm{out}}, I_u^{\mathrm{out}})$ independently of the input label $(S_u^{\mathrm{in}}, I_u^{\mathrm{in}})$. The unit can then declassify/endorse parts with tag t (cf. §3.1.4).
changeInOutLabel($\langle \mathrm{S} \vert \mathrm{I} \rangle$, $\langle \mathrm{add} \vert \mathrm{del} \rangle$, t)	Adds/removes tag t to/from a unit's input label and output label.

Table 1: Description of the DEFCON API available to event processing units. Note that due to contamination independence S and I in API calls may be transparently changed by the system: $S' = S \cup S_u^{\mathrm{out}}$ and $I' = I \cap I_u^{\mathrm{out}}$

Manual inspection. JDK methods that synchronise on locks cannot safely be accessed from units. For example, `Classloader.loadClass()` and many `StringBuffer` methods are synchronised. However, both are types that are never shared, i.e. they satisfy the above three requirements. Instead of modifying them in the JDK source-code, we transformed them to implement `NeverShared` through an aspect that is applied before the interception aspect.

5 DEFCON API

We built a DEFCON prototype system in Java that implements the DEFC model and enforces isolation as described in §4. The API calls that units may use to interact with the DEFCON system are described in Table 1.

Contamination independence. Most of the calls do not impose restrictions on the caller, yet they are safe because of a unit's contamination. Calls such as `addPart()`, which adds a new part to an event (cf. Table 1), should not fail if a unit is unable to write at the requested contamination level because units may not be aware of their initial contamination. Instead DEFCON guarantees that any tags present in the unit's current output label are attached transparently to generated parts. For example, a unit with a label $S_u^{\mathrm{out}} = \{d\}$ that invokes `addPart` with label $S = \{t\}$ causes that part to be labelled $S' = \{d, t\}$. This highlights an important property of the API: *contamination independence*. It allows a unit to be sandboxed by instantiating it at a higher contamination level that it is unaware of. All of its input and output will be affected by this initial contamination.

Freezing shared objects. Most of the API calls receive or return potentially mutable objects. References to these objects may not be communicated to other units since changes to their state cannot be controlled. In particular, this applies to objects representing event parts and labels.

The `addPart()` call allows a unit to include objects of various types in a part. For immutable types, making shared references is safe. However, this is not true for mutable types (e.g. `Date`) or collection types that support adding multiple objects to a part (e.g. `HashMap<Date>`). To avoid the cost of serialising and copying such types during event dispatching, DEFCON limits contents of event parts to a subset of types. These types must be either immutable or extend a package-private `Freezable` base class.

For `Freezable` objects, mutating operations are disallowed after a call to `freeze()` has been made. This incurs the overhead of checking an `isFrozen` flag on each mutating operation. For collection classes, a call to `freeze()` must efficiently freeze all contained objects. To avoid iterating through collections, each `Freezable` object that is attached to a `Freezable` collection has a reference to the collection's `isFrozen` flag. This makes `freeze()` a constant time operation. The overhead of mutating operations on a `Freezable` object is linear with the number of collections the object is part of. A similar approach is used to make the `Label` type immutable.

6 Evaluation

The goal of our experimental evaluation is to demonstrate the practicality of the DEFC model in high-performance event processing. We describe the implementation and evaluation of a simple financial stock trading platform built using DEFCON. We compare our implementation to *Marketcetera* [3], a popular open source trading platform written in Java. Although only one of few open source offerings in a space dominated by proprietary solutions, Marketcetera is gaining momentum by providing performance comparable to proprietary systems [35] and features such as rapid strategy development, complex event processing and interaction with various exchanges.

We quantify event processing performance using two metrics: *event throughput*, the number of events processed per unit time, and *event latency*, the delay that events experience when being processed. We also measure the overhead of our DEFC approach on event-driven applications in terms of processing performance and memory consumption.

DEFCON isolates the trading strategies of traders, thus allowing multiple strategies to be hosted on the same machine as the stock market feed. Instead, Marketcetera uses multiple JVMs, one per client, to isolate trading strategies, limiting the scalability of a similar deployment to a smaller number of traders. DEFCON achieves low event processing latency and high throughput, while scaling to 10 times the number of traders compared to Marketcetera. As explained in §2.1, this makes co-location affordable to more traders because a single machine can securely serve a larger number of trading strategies.

6.1 Financial trading scenario

We adopt a classic trading algorithm called *pairs trade* [39]. It is based on the observation that changes in the stock prices of established companies in the same industry are frequently correlated. Traders use this to predict the stock price for the immediate future and gain a return. The performance of the pairs trade algorithm depends heavily on the latency of trades: co-located traders' orders will increase the price of the cheapest stock and decrease the most expensive one, thus limiting the profit margins for remote traders.

Marketcetera implementation. We implemented the pairs trading strategy as a Strategy Agent in Marketcetera 1.5.0. Strategy Agents host one or more strategies of the same client. For isolation, a separate JVM is created for each client's Strategy Agent. Since third-party libraries may leak client data that they receive, each client must vouch for their strategy's implementation or fully trust its developer. To use brokering services, Strategy Agents communicate with an Order Routing Service (ORS) that forwards requests to an exchange. We extended the ORS to provide local brokering facilities and a corresponding market data feed to the Strategy Agents.

DEFCON implementation. DEFCON allows competing traders to execute latency-sensitive operations in the same JVM. The confidentiality guarantees provided by the DEFC model enable concurrent execution of proprietary trading algorithms without fear of disclosure. As illustrated in Figure 4, our trading platform has the following processing units:

Trader units encapsulate traders' strategies for buying and selling stocks using pairs trading.

Pair Monitor units provide pairs trading as a service since it is used by all traders in our system. Based on a stock pair and an investment threshold, it sends events to traders when the expected price difference occurs.

A **Local Broker unit** enables traders to clear their orders locally, without the need to involve the stock exchange, by matching traders' bid/ask orders.

A **Stock Exchange unit** is responsible for the communication with the stock exchange. In its simplest form, it is the source of events regarding trades that occur there.

A **Regulator unit** samples a subset of local trades on behalf of a regulatory body. It may verify that the volume of a trader's trades has not exceeded a given quota.

DEFCON operation. We describe the operation of our trading platform using steps 1–9 in Figure 4.

Step 1: A Trader unit declares its interest in a given pair of stocks. The published event is protected by a unique trader tag t_1, owned by Trader 1. Since the selection of stocks and the parameters of the pairs trade are sensitive information that must be protected from disclosure, Trader 1 delegates the t_1^{+} privilege only to the corresponding Pair Monitor unit. This Pair Monitor uses this privilege to learn about the pair of symbols to monitor. All its output will only be visible to the Trader 1 that owns t_1.

Step 2: The Pair Monitor issues two tick event subscriptions: one for each of the two symbols that it has to mon-

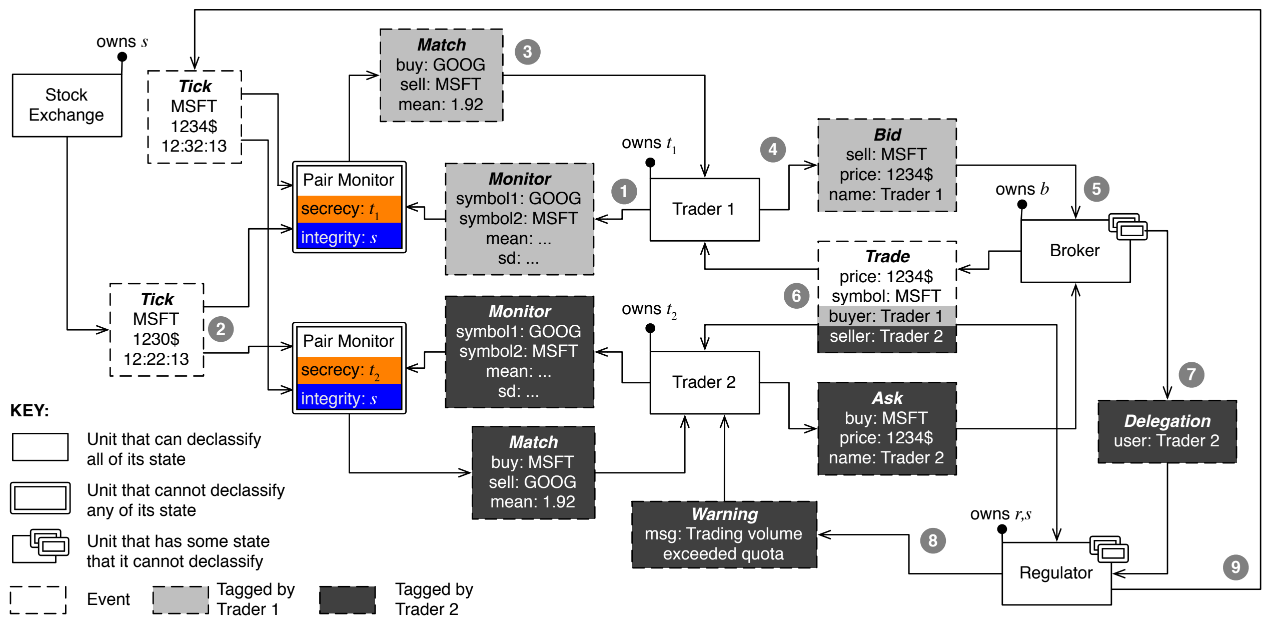


Figure 4: Workflow of the implementation of our stock trading platform in DEFCon, highlighting the DEFC aspects.

itor. Pair Monitor units are always instantiated with read integrity s and are thus only able to perceive events published by the Stock Exchange unit that owns s.

Step 3: Once a tick event is published with an adequate price, the Pair Monitor sends an event to the Trader. This event is tagged with t_1 and Trader 1 is the only unit with the necessary confidentiality read label to receive it.

Step 4: Trader 1 may decide to sell stocks using the local brokering facilities. A bid order is generated with the offered price and the issuing Trader's details. The issuing Trader has three different security requirements for a published order: (1) it must convey the details of the order to the Broker to be matched against other orders; (2) no other unit, apart from the Broker, should be able to associate the order with the issuing Trader; (3) no unit should be able to correlate two orders by the same Trader. A competing Trader correlating orders may learn the underlying trading strategy. In addition, a Trader does not trust the Broker not to reveal information about trades. To capture these security requirements, the first part of the bid, price/symbol, is protected by a broker tag b while the second, name, is protected both by b and by a randomly-generated tag t_r used only for this order. The Broker has b^+ and b^-. The first part also carries the privilege t_r^+, allowing only the Broker to see the Trader's name as long as it accepts the additional contamination.

Step 5: The Broker can read the first part and declassify it. It has to use a *managed subscription* (cf. §5) to learn the Trader's identity. Once a corresponding ask order is received, the Broker matches it and completes a trade.

Step 6: The first part of the trade event is declassified and visible publicly. The two additional parts with the sensitive identities of the Traders are protected individually by unique tags. Each Trader can identify its own trades while DEFCon guarantees that no other unit can do the same.

Step 7: A Regulator may intercept trades to verify that they are compliant with trading rules. If it observes a suspicious trade, it uses a managed subscription to receive t_r^+ over the unique tag t_r that protects the Trader's identity. Since this privilege is only needed for suspicious trades, the Regulator receives it from the Broker on-demand. The delegation event is only possible as long as $t_r^{+\text{auth}}$ was included in the second part of the bid order in step 4.

Steps 8+9: With this privilege, the Regulator can communicate a warning to the Trader. The Regulator owns the integrity tag s and is able to republish the local trade as a valid stock tick perceivable by the Pair Monitors.

6.2 Experimental results

We evaluated our system with a synthetic workload of stock tick events that was derived from traces of trades made on the London Stock Exchange. In our workload, we selected the tick prices so that they triggered the pairs trading algorithm for each pair once every 10 ticks. This approach both generated a significant order load and also allowed us to avoid the issue of choosing suitably correlated pairs from real market data. Since the main bottleneck was the filtering that occurred between the Stock Exchange and the Pair Monitor units, the tick rate achieved only caused transient queuing in the system.

We varied the number of Traders on the platform. Each Trader monitors a single symbol pair that was chosen according to a Zipf distribution. This emulated the fact that some symbol pairs are well known to be correlated and, as a result, the majority of Traders monitor their prices. All tests were run on a dual processor Intel Xeon E5540 2.53 GHz machine with a maximum of 1 GiB heap memory using Sun's Hotspot JVM, version 1.6.0_16.

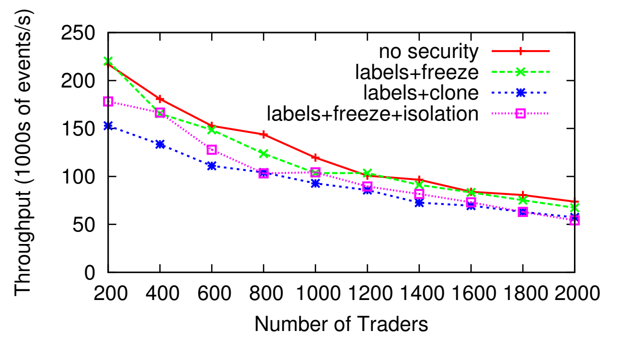


Figure 5: Maximum supported event rate in DEFCon as a function of the number of traders.

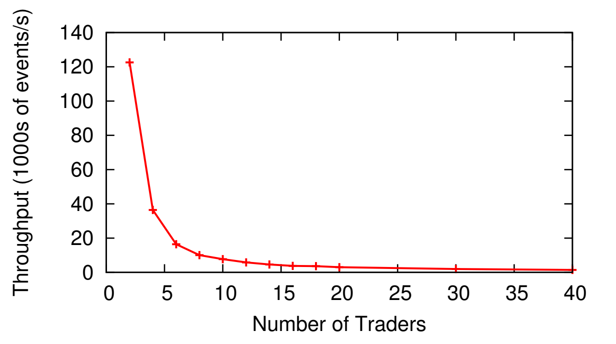


Figure 8: Maximum supported event rate in Marketcetera as a function of the number of traders.

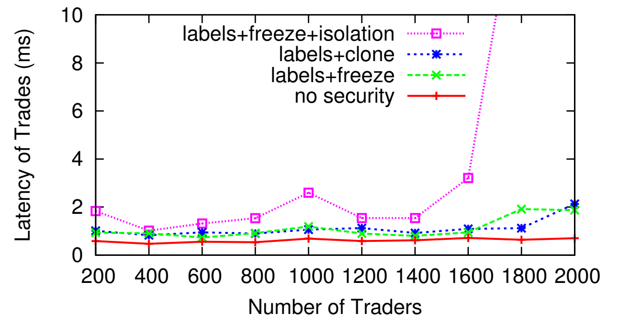


Figure 6: Event processing latency in DEFCon as a function of the number of traders.

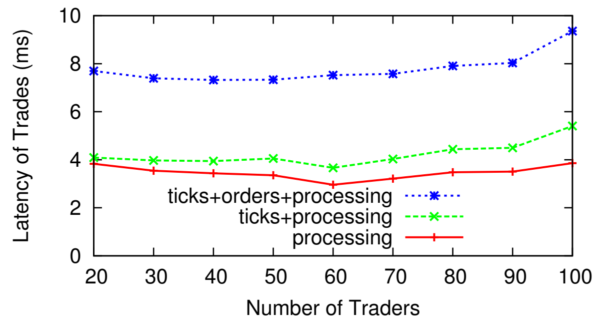


Figure 9: Event processing latency in Marketcetera broken down into individual contributions.

DEFCon performance. To explore the limits of our DEFCon deployment, we had the Stock Exchange unit replay tick event traces as quickly as possible, while measuring the achieved throughput every 100 ms. Figure 5 shows the median throughput when increasing the number of Traders in the system. In the simplest case without security (*no security*), the system performance ranges from 220,000 events per second with 200 Traders to 75,000 events with 2,000 Traders. (Note that the Stock Exchange unit in our implementation is single-threaded.) The overhead of introducing labels and freezable objects (*labels+freeze*) is within the error margin, while the overhead of cloning (*labels+clone*) is around 30%, even with the simple data structures of our financial application. The overhead of adding isolation (*labels+freeze+isolation*) is around 20%, staying constant with the number of Traders.

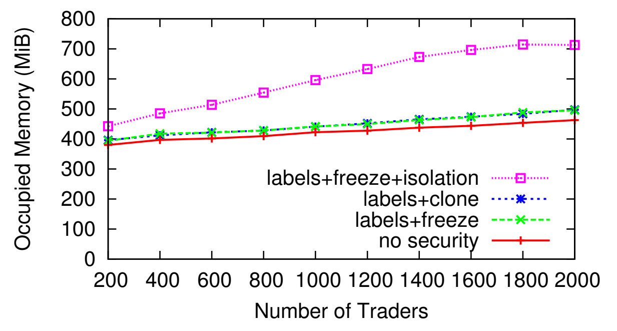


Figure 7: Amount of used memory in DEFCon as a function of the number of traders.

Next we measured latency as the time difference between when a trade event is produced by the Broker and the time when the originating tick event occurred. This includes the processing time of the Stock Exchange, Pair Monitor, Trader and Broker units. In Figure 6, we plot the 70th percentile of latencies, again increasing the number of Traders. We ignore higher latency percentiles because they are affected by the characteristics of the workload and the operation of the Java garbage collector. Spikes in trading activity, as commonly found when markets open, result in transient congestion in the Broker and thus queueing of events. Short periodic activations of the garbage collector preempt processing threads for about 20 ms and increase the latency of individual events.

Figure 6, shows that the latency without security (*no security*) is about 0.5 ms independently of the number of Traders. Introducing label checks into the system sees latencies rise to approximately 1 ms without and 2 ms with isolation. This behaviour continues up to 1,500 Traders after which the system becomes overloaded.

In Figure 7, we measured the memory consumption in the above experiment. Of the total memory consumed, about 300 MiB is used to cache tick events. We observe that while the overhead of (*labels+freeze*) is minimal compared to the base case, the weaving framework incurs an overhead of 50 MiB for 200 Traders, and up to 200 MiB for 2,000 Traders.

Marketcetera performance. We compare the results from DEFCON with the performance of Marketcetera. The median throughput of Marketcetera is shown in Figure 8. Although the event rate for only 2 Traders is high, the system does not scale well. (Note the lower number of Traders compared to Figure 5.) With just 10 active Traders, the throughput falls below 10,000 events per second. This is mostly due to Strategy Agents filtering market data individually as the platform does not support centralised market data filtering. Memory consumption is also significantly higher in Marketcetera. Starting from 2 GiB for 20 Traders, the used memory across all JVMs reaches 6 GiB for 100 Traders. Without multiple JVMs allocating memory, DEFCON manages to support 1,500 Traders using less than 1 GiB of heap space.

For measuring latency in Marketcetera, we chose a low rate of 1,000 events per second in the stock feed with a small number of Traders. This reduces the CPU load and allows us to draw conclusions about latency while not being affected by scheduling phenomena. In Figure 9, we show the 70th percentile of trades' latencies in Marketcetera, measured at the broker, when increasing the number of Traders. (Figure 6 shows the corresponding result for DEFCON but again note the lower number of Marketcetera Traders.) Latency in Marketcetera is around 8 ms. The plot breaks this total latency down into its individual contributions: the time to filter unwanted events and execute the pairs trading algorithm (*processing*), the time for tick propagation from the Market Feed to the Strategy Agents (*ticks+processing*) and order propagation from Strategy Agents to the ORS (*ticks+orders+processing*). When we introduced 100 Traders, the increasing cost of communication across JVMs surpassed the actual processing latency. In contrast, DEFCON is able to provide latency at around 1 ms for significantly more Traders. We believe that this is because DEFCON tick propagation uses our event dispatching mechanism, which does not involve communication across JVMs.

Security comparison. In DEFCON, the pairs trading algorithm is a service that each Trader may decide to use. DEFC guarantees that the unit that implements the algorithm does not have the ability to leak traders' choices. Marketcetera requires that each Strategy implementation be fully-trusted. As a result, it does not support third-party services that Strategies may use. Moreover, DEFCON enables the efficient reuse of a single trading strategy across multiple users; Marketcetera instead requires a new JVM each time. Code reuse is particularly beneficial when traders belong to the same organisation such as a small hedge fund. The Marketcetera Broker can, deliberately or involuntarily, leak a user's trades or orders to other parties. In contrast, the DEFCON Broker is prevented by DEFC from correlating two clients' orders. The Broker's developers can guarantee that no bugs may result in data leaks that violate the Traders' confidentiality. In addition, a regulatory service can only reliably be integrated into Marketcetera by its original developers. Instead, DEFCON can support a Regulator unit without concern that the additional code may damage the security properties of the system.

7 Conclusions

High-performance event processing applications, for example as found in algorithmic stock trading, need strong information security without sacrificing performance. We presented DEFCON: an event processing system that enforces *decentralised event flow control* (DEFC). This model meets the particular security needs of event processing by providing mandatory protection of event data from bugs and intentional leaks. DEFCON relies on isolation at the programming language level and we described a practical methodology for achieving isolation in Java with low manual effort. By isolating processing units running in the same address space, we tried to strike an optimal balance between the need for isolation and efficient inter-isolate communication. Our evaluation shows the practicality of our solution when compared to an open source trading platform.

In future work, we plan to investigate issues in a distributed system build from a set of DEFCON nodes. We also want to explore additional techniques for isolation in Java, such as dynamic recompilation of classes, and approaches that facilitate provably secure hand-off of Java references, such as Kilim [36]. Although we do not address denial-of-service attacks in this work, we believe that thanks to our message passing paradigm it is possible to use common profiling techniques from aspect-oriented programming for resource accounting [40]. In related work, we have begun to investigate how the DEFC model can be applied to functional languages such as Erlang that naturally support isolation between components through message-passing [31].

Acknowledgements

We thank Oli Bage, Eva Kalyvianaki, Cristian Cadar and Jonathan Ledlie for comments on earlier paper drafts. We also thank our anonymous reviewers and our shepherd, Nickolai Zeldovich, for their guidance. This work was supported by grants EP/F042469 and EP/F044216 ("SmartFlow: Extendable Event-Based Middleware") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] Aite Group. Market data infrastructure challenges, April 2009. www.aitegroup.com/reports/200904222.php.

[2] Progress Apama Website. www.progress.com/apama.

[3] Asay, M. Marketcetera gives hedge funds cloud-based trading. *Cnet News* (2009).

[4] Back, G., Hsieh, W. C., and Lepreau, J. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *OSDI'00* (San Diego, CA, USA), USENIX, pp. 23–23.

[5] Beeri, C., Eyal, A., Milo, T., and Pilberg, A. Monitoring business processes with queries. In *VLDB'07* (Vienna, Austria).

[6] Bell, D. E., and La Padula, L. J. Secure computer systems: Mathematical foundations and model. Tech. Rep. M74-244, The MITRE Corp., Bedford, MA, USA, May 1973.

[7] Betfair website. www.betfair.com.

[8] Brewer, D., and Nash, M. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy* (Oakland, CA, USA, May 1989).

[9] Clark, J. Still the need for speed. *Waters* (2008).

[10] Czajkowski, G., and Daynes, L. Multitasking without compromise: A virtual machine evolution. In *OOPSLA'01*.

[11] Department of Defense. Trusted computer system evaluation criteria (Orange Book), 1983.

[12] Duhigg, C. Stock traders find speed pays, in milliseconds. *The New York Times* (2009).

[13] Efstathopoulos, P., Krohn, M., VanDeBogart, S., et al. Labels and event processes in the Asbestos Operating System. In *SOSP '05* (Brighton, UK), ACM, pp. 17–30.

[14] Esper website. esper.codehaus.org.

[15] Financial Service Authority. Press releases, 1997-2009. available at www.fsa.gov.uk.

[16] Flatley, R. Check your speed. *The Trade News* (2007).

[17] Geoffray, N., Thomas, G., Muller, G., et al. I-JVM: a Java virtual machine for component isolation in OSGi. In *Dependable Systems and Networks (DSN)* (Estoril, Portugal, April 2009), p. 10.

[18] Golm, M., Felser, M., Wawersich, C., and Kleinöder, J. The JX Operating System. In *USENIX ATC'02* (Monteray, CA, USA), pp. 45–58.

[19] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. Implementing multiple protection domains in Java. In *USENIX ATC'98* (New Orleans, LA, USA).

[20] Iati, R. The real story of trading software espionage. *Advanced Trading* (2009).

[21] Kiczales, G., Lamping, J., and others, A. M. Aspect-oriented programming. In *ECOOP'97* (Jyvskyl, Finland).

[22] Krohn, M., Yip, A., Brodsky, M., et al. Information flow control for standard OS abstractions. In *SOSP'07* (Stevenson, WA, USA), ACM, pp. 321–334.

[23] London Stock Exchange. Hosting capacity increases fivefold. Press Release, November 2009.

[24] Luckham, D. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[25] Mettler, A., Wagner, D., and Close, T. Joe-E: A security-oriented subset of Java. In *Network and Distributed System Security (NDSS)* (Dan Diego, CA, USA, 2010), Internet Society.

[26] Mühl, G., Fiege, L., and Pietzuch, P. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[27] Myers, A., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology 9*, 4 (2000), 410–442.

[28] Myers, A. C. JFlow: Practical mostly-static information flow control. In *POPL'99* (San Antonio, TX, USA).

[29] Nair, S., Simpson, P., et al. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science 197*, 1 (2008), 3–16.

[30] Palacz, K., Vitek, J., Czajkowski, G., and Daynas, L. Incommunicado: efficient communication for isolates. In *OOPSLA'02* (Seattle, WA, USA), ACM, pp. 262–274.

[31] Papagiannis, I., Migliavacca, M., Eyers, D. M., Shand, B., Bacon, J., and Pietzuch, P. Enforcing user privacy in web applications using Erlang. In *Web 2.0 Security and Privacy (W2SP)* (Oakland, CA, USA, 2010), IEEE.

[32] Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. Laminar: practical fine-grained decentralized information flow control. In *PLDI'09* (Dublin, Ireland), ACM.

[33] Schanzer, E. Performance considerations for run-time technologies in the .NET framework. msdn.microsoft.com/en-us/library/ms973838.aspx, Aug 2001.

[34] Schmerken, I. Skyrocketing market data message rates leading trading firms to consider hardware acceleration. *Wall Street & Technology* (2007).

[35] Shoemaker, K. Marketcetera 1.5 release note. *Ostatic* (2009).

[36] Srinivasan, S., and Mycroft, A. Kilim: Isolation-typed actors for Java. In *ECOOP'08* (Paphos, Cyprus).

[37] Stonebraker, M., Çetintemel, U., and Zdonik, S. The 8 requirements of real-time stream processing. *SIGMOD Rec. 34*, 4 (2005), 42–47.

[38] Thome, B., Gawlick, D., and Pratt, M. Event processing with an Oracle database. In *SIGMOD'05* (Baltimore, MD), ACM.

[39] Vidyamurthy, G. *Pairs Trading: Quantitative Methods and Analysis*. Wiley Finance, 2004.

[40] Villazón, A., Binder, W., and Moret, P. Aspect weaving in standard Java class libraries. In *Principles and Practice of Programming in Java (PPPJ)* (Modena, Italy, 2008), ACM.

[41] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving application security with data flow assertions. In *SOSP'09* (Big Sky, MT, USA), ACM, pp. 291–304.

[42] Zeldovich, N., Boyd-Wickizer, S., and Mazières, D. Securing distributed systems with information flow control. In *NSDI'08* (San Francisco, CA, USA), pp. 293–308.

[43] Zeldovich, N., Kohler, E., et al. Making information flow explicit in HiStar. In *OSDI'06* (Seattle, WA, USA), USENIX Association, pp. 19–19.

[44] Zendrian, A. Don't be afraid of the dark pools. *Forbes* (May 2009).

Wide-Area Route Control for Distributed Services

Vytautas Valancius*, Nick Feamster*, Jennifer Rexford†, Akihiro Nakao‡
*Georgia Tech †Princeton University ‡The University of Tokyo

ABSTRACT

Many distributed services would benefit from control over the flow of traffic to and from their users, to offer better performance and higher reliability at a reasonable cost. Unfortunately, although today's cloud-computing platforms offer elastic computing and bandwidth resources, they do not give services control over wide-area routing. We propose replacing the data center's border router with a *Transit Portal* (TP) that gives each service the illusion of direct connectivity to upstream ISPs, without requiring each service to deploy hardware, acquire IP address space, or negotiate contracts with ISPs. Our TP prototype supports many layer-two connectivity mechanisms, amortizes memory and message overhead over multiple services, and protects the rest of the Internet from misconfigured and malicious applications. Our implementation extends and synthesizes open-source software components such as the Linux kernel and the Quagga routing daemon. We also implement a management plane based on the GENI control framework and couple this with our four-site TP deployment and Amazon EC2 facilities. Experiments with an anycast DNS application demonstrate the benefits the TP offers to distributed services.

1. Introduction

Cloud-based hosting platforms make computational resources a basic utility that can be expanded and contracted as needed [10, 26]. However, some distributed services need more than just computing and bandwidth resources—they need control over the *network*, particularly over the wide-area routes to and from their users. More flexible route control helps improve performance [7, 8, 12] and reduce operating costs [17]. For example, interactive applications like online gaming want to select low-latency paths to users, even if cheaper or higher-bandwidth paths are available. As another example, a service replicated in multiple locations may want to use IP anycast to receive traffic from clients and adjust where the address block is announced in response to server failures or shifts in load.

Although flexible route control is commonplace for both content providers and transit networks, today's cloud-based services do not enjoy the same level of control over routing. Today, the people offering these kinds of distributed services have two equally unappealing options. On the one hand, they could build their own network footprint, including acquiring address space, negotiating contracts with ISPs, and installing and configuring routers. That is, they could essentially become network operators, at great expense and with little ability to expand their footprint on demand. On the other hand, they could contract with a hosting company and settle for whatever "one size fits all" routing decisions this company's routers make.

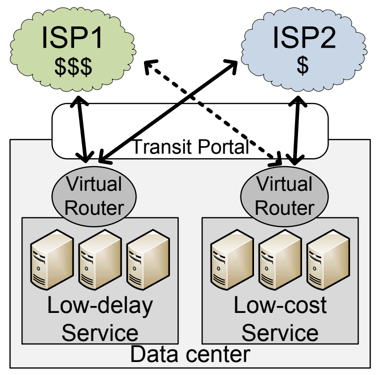


Figure 1: Connecting services though the Transit Portal.

This missed opportunity is not for a lack of routing diversity at the data centers: for example, RouteViews shows that Amazon's Elastic Cloud Computing (EC2) has at least 58 upstream BGP peers for its Virginia data center and at least 17 peers at its Seattle data center [20]. Rather, cloud services are stuck with a "one size fits all" model because cloud providers select a single best route for all services, preventing cloud-based applications from having any control over wide-area routing.

To give hosted services control over wide-area routing, we propose the *Transit Portal (TP)*, as shown in Figure 1. Each data center has a TP that gives each service the appearance of direct connectivity to the ISPs of its choice. Each service has a dedicated *virtual router* that acts as a gateway for the traffic flowing to and from its servers. The service configures its virtual router with its own policies for *selecting paths* to its clients and *announcing routes* that influence inbound traffic from its clients. By offering the abstraction of BGP sessions with each upstream ISP, the TP allows each service to capitalize on existing open-source software routers (including simple lightweight BGP daemons) without modifying its application software. That said, we believe extending TP to offer new, programmatic interfaces to distributed services is a promising avenue for future work.

Using the TP to control routing provides a hosted service significantly more control over the flow of its traffic than in today's data centers. In addition, the services enjoy these benefits without building their own network footprint, acquiring address space and AS numbers, and negotiating with ISPs. These are hurdles that we ourselves faced in deploying TPs at several locations; the TP obviates the need for the services that it hosts to do the same. In addition, the TP simplifies operations for the ISPs by offloading the separate connections and relationships with each application and by applying packet and route filters to protect them (and the rest of the Internet) from misconfigured or malicious services.

The design and implementation of the TP introduces several challenges. In the control plane, the TP must provide each virtual router the appearance of direct BGP sessions to its upstream ISPs. The TP must also forward outgoing packets to the right ISP and demultiplex incoming packets to the right virtual router. Our solutions to these problems must scale as the number of services increases. To solve these problems, we introduce a variety of techniques for providing layer-two connectivity, amortizing memory and message overhead, and filtering packets and routes. Our prototype implementation composes and extends open-source routing software—the Quagga software router for the control plane and the Linux kernel for the data plane—resulting in a system that is easy to deploy, maintain, and evolve. We also built a management system based on the GENI control framework [16] that automates the provisioning of new customers. The TP is deployed and operational at several locations.

This paper makes the following contributions:

- We explain how flexible wide-area route control can extend the capabilities of existing hosting platforms.
- We present the design and implementation of Transit Portal and demonstrate that the system scales to many ISPs and clients.
- We quantify the benefits of TP by evaluating a DNS service that uses IP anycast and inbound traffic engineering on our existing TP deployment.
- We present the design and implementation of a management framework that allows hosted services to dynamically establish wide-area connectivity.
- We describe how to extend the TP to provide better forwarding performance and support a wider variety of applications (*e.g.*, virtual machine migration).

The remainder of the paper is organized as follows. Section 2 explains how distributed services can make use of wide-area route control. Section 3 presents the design and implementation of the Transit Portal. Section 4 evaluates our three-site deployment supporting an example service, and Section 5 evaluates the scalability and performance of our prototype. Section 6 presents our management framework, and Section 7 describes possible extensions to the TP. Section 8 compares TP to related work; we conclude in Section 9.

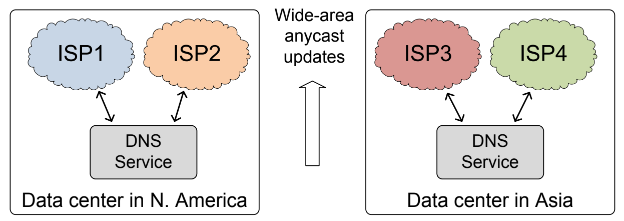


Figure 2: ***Reliable, low-latency distributed services:*** **A service provider that hosts authoritative DNS for some domain may wish to provision both hosting and anycast connectivity in locations that are close to the clients for that domain.**

2. A Case for Wide-Area Route Control

We aim to give each hosted service the same level of routing control that existing networks have. Each service has its own virtual router that connects to the Internet through the *Transit Portal*, as shown in Figure 1. The Transit Portal allows each service to make a different decision about the best way to exchange traffic with its users. The Transit Portal also allows each service to announce prefixes selectively to different ISPs or send different announcements for the same IP prefix to control how inbound traffic reaches downstream networks.

2.1 How Route Control Helps Applications

This section describes three services that can benefit from having more control over wide-area routing and the ability to rapidly provision connectivity. Section 4 evaluates the first service we discuss—improving the reliability, latency, and load balacing traffic for distributed services—through a real deployment on Amazon's EC2. We do not evaluate the remaining applications with a deployment, but we explain how they might be deployed in practice.

Reliable, low-latency distributed services. The Domain Name System (DNS) directs users to wide-area services by mapping a domain name to the appropriate IP address for that service. Service providers often use DNS for tasks like load balancing. Previous studies have shown that DNS lookup latency is a significant contributor to the overall latency for short sessions (*e.g.*, short HTTP requests). Thus, achieving reliability and low latency for DNS lookups is important. One approach to reducing DNS lookup latency is to move the authoritative DNS servers for a domain closer to clients using anycast. Anycast is a method where multiple distinct networks advertise the same IP prefix; client traffic then goes to one of these networks. Hosting authoritative name servers on an anycasted IP prefix can reduce the round-trip time to an authoritative name server for a domain, thus reducing overall name lookup time.

Although anycast is a common practice for DNS root servers, setting up anycast is a tall order for an individ-

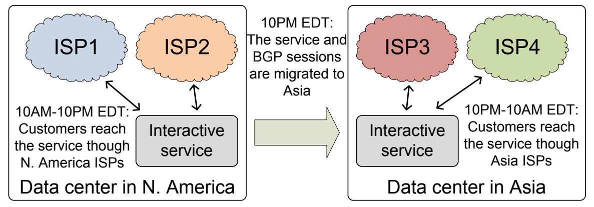

ISP1
ISP2
10AM-10PM EDT: Customers reach the service though N. America ISPs
Interactive service
Data center in N. America
10PM EDT: The service and BGP sessions are migrated to Asia
ISP3
ISP4
Interactive service
10PM-10AM EDT: Customers reach the service though Asia ISPs
Data center in Asia

Figure 3: ***Using routing to migrate services:*** **A service provider migrates a service from a data center in North America to one in Asia, to cope with fluctuations in demand. Today, service providers must use DNS for such migration, which can hurt user performance and does not permit the migration of a running service. A provider can use route control to migrate a service and re-route traffic on the fly, taking DNS out of the loop and enable migration of running services.**

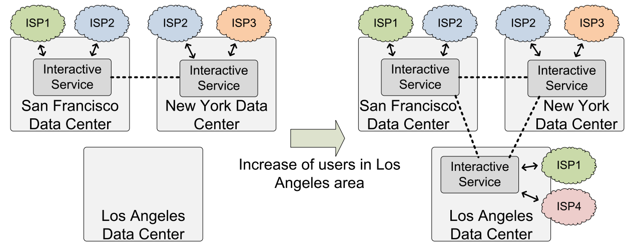


Figure 4: ***Flexible peering and hosting for interactive applications:*** **Direct control over routing allows services to expand hosting** ***and upstream connectivity*** **in response to changing demands. In this example, a service experiences an increase in users in a single geographic area. In response, it adds hosting and peering at that location to allow customers at that location to easily reach the service.**

ual domain: each domain that wants to host its own authoritative servers would need to establish colocation and BGP peering at multiple sites and make arrangements. Although a DNS hosting provider (*e.g.*, GoDaddy) could host the authoritative DNS for many domains and anycast prefixes for those servers, the domains would still not be able directly control their own DNS load-balancing and replication. Wide-area route control allows a domain to establish DNS-server replicas and peering in multiple locations and to change those locations and peering arrangements when load changes. Figure 2 shows such a deployment. We have deployed this service [24] and will evaluate it in Section 4.

Using routing to migrate services. Service providers such as Google commonly balance client requests across multiple locations and data centers to keep the latency for their services as low as possible. To do so, they commonly use the DNS to re-map a service name to a new IP address. Unfortunately, relying on DNS to migrate client requests requires the service provider to set low time-to-live (TTL) values on DNS replies. These low TTL values help a service provider quickly re-map a DNS name to a new IP address, but they also prevent the client from caching these records and can introduce significant additional latency; this latency is especially troublesome for short-lived sessions like Web search, where the DNS lookup comprises a large fraction of overall response time. Second, DNS-based re-mapping cannot migrate ongoing connections, which is important for certain services that maintain long-lived connections with clients (*e.g.*, VPN-based services). Direct wide-area route control allows the application provider to instead migrate services using routing: providers can migrate their services without changing server IP addresses by dynamically acquiring wide-area connections and announcing the associated IP prefix at the new data center while withdrawing it at the old one. Figure 3 shows how this type of migration can be implemented. This approach improves user-perceived performance by allowing the use of larger DNS TTL values and supporting live migration of long-lived connections.

Flexible peering & hosting for interactive applications. To minimize round-trip times, providers of interactive applications like gaming [2] and video conferencing [3, 4] aim to place servers close to their customers to users and, when possible, selecting the route corresponding to the lowest-latency path. When traffic patterns change due to flash-crowds, diurnal fluctuations, or other effects, the application provider may need to rapidly reprovision both the locations of servers *and* the connectivity between those servers and its clients. Figure 4 shows an example of an interactive service that suddenly experiences a surge in users in a particular region. In this case, the hosting facility will not only need to provision additional servers for the interactive service provider, but it will also need to provision additional connectivity at that location to ensure that traffic to local clients enter and leave at that facility.

2.2 Deployment Scenarios

Cloud providers can provide direct control over routing and traffic to hosted applications. A cloud service such as Amazon's EC2 can use direct wide-area route control to allow each application provider to control inbound and outbound traffic according to its specific requirements. Suppose that two applications are hosted in the same data center. One application may be focused on maintaining low-cost connectivity, while the other may want to achieve low latency and good performance at any cost. Today's cloud services offer only "one size fits all" transit and do not provide routing control to each hosted service or application; the Transit Portal provides this additional control.

An enterprise can re-provision resources and peering as demands change. Web service providers such as Google and Microsoft share a common infrastructure across multiple applications (*e.g.*, search, calendar, mail, video) and continually re-provision these resources as client demand shifts. Today, making application-specific routing decisions in a data center (as shown in Figure 1) is challenging, and re-routing clients to services in different data centers when demands change is even more difficult.

The Transit Portal can provide each application in a data center control over routing and peering, allowing it to establish connectivity and select paths independently of the other properties. This function also makes service migration easier, as we describe in further detail below.

A researcher can perform experiments using wide-area routing. Although existing testbeds such as Emulab [14] allow researchers to operate their own wide-area networks, they generally do not offer flexible control over connectivity to the rest of the Internet. Different experiments will, of course, have different requirements for the nature of their connectivity and routing, and researchers may even want to experiment with the effects of different peering arrangements on experimental services. As part of the GENI project, we are building facilities for this level of route control by connecting Transit Portal to downstream virtual networks to allow researchers to design and evaluate networked services that require greater control over wide-area routing.

3. Design and Implementation

This section describes the design and implementation of a Transit Portal (TP); Section 6 completes the picture by describing the management framework for a network of TPs. The TP extends and synthesizes existing software systems—specifically, the *Linux* kernel for the data plane and the *Quagga* routing protocol suite for the control plane. The rest of this section describes how our design helps the TP achieve three goals: (1) *transparent connectivity* between hosted services and upstream ISPs; (2) *scalability* to many hosted services and upstream ISPs; and (3) the ability to *protect* the rest of the Internet from accidental or malicious disruptions. Table 1 summarizes our design and implementation decisions and how they allow us to achieve the goals of transparent connectivity, scalability, and protection.

3.1 Transparent Connectivity

The TP gives client networks the appearance of direct data- and control-plane connectivity to one or more upstream ISPs. This transparency requires each client network to have a layer-two link and a BGP session for each upstream ISP that it connects to, even though the link and session for that client network actually terminate at the TP. The client's virtual routers are configured exactly as they would be if they connected directly to the upstream ISPs without traversing the Transit Portal. The TP has one layer-two connection and BGP session to each upstream ISP; this connection multiplexes both data packets and BGP messages for the client networks.

Different layer-two connections for different clients. Connecting to an upstream ISP normally requires the client to have a direct layer-two link to the ISP for carrying both BGP messages and data traffic. To support this

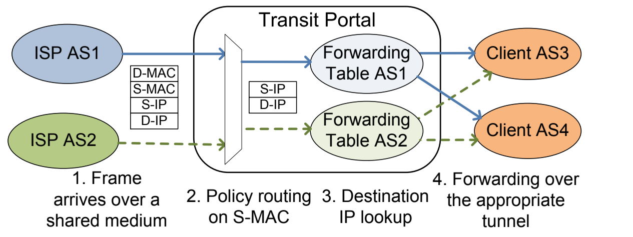


Figure 5: *Forwarding incoming traffic:* When a packet arrives at the Transit Portal (Step 1), the TP uses the source MAC address (S-MAC) to demultiplex the packet to the appropriate IP forwarding table (Step 2). The lookup in that table (Step 3) determines the appropriate tunnel to the client network (Step 4).

```
1  # arp -a
2  router1.isp.com (1.1.1.1) at 0:0:0:0:0:11 on
       eth0
3  # iptables -A PREROUTING -t mangle -i eth0 -m
       mac --mac-source 0:0:0:0:0:11 -j MARK
       --set-mark 1
4  # ip rule add fwmark 1 table 1
```

Figure 6: Linux policy routing de-multiplexes traffic into the appropriate forwarding table based on the packet's source MAC address. In this example, source address `0:0:0:0:0:11` de-multiplexes the packet into forwarding table `1`.

abstraction, the TP forms a separate layer-two connection to the client for each upstream ISP. Our implementation uses the Linux 2.6 kernel support for IP-IP tunnels, GRE tunnels, EGRE tunnels, and VLANs, as well as UDP tunnels through a user-space OpenVPN daemon.

Transparent forwarding between clients and ISPs. The TP can use simple policy routing to direct traffic from each client tunnel to the corresponding ISP. Forwarding traffic from ISPs to clients, however, is more challenging. A conventional router with a single forwarding table would direct the traffic to the client prefix over a single link (or use several links in a round robin fashion if multipath routing is enabled.) As shown in Figure 5, though, the TP must ensure the packets are directed to the appropriate layer-two link—the one the client's virtual router associates with the upstream ISP. To allow this, the TP maintains a *virtual forwarding table* for each upstream ISP. Our implementation uses the Linux 2.6 kernel's support for up to 252 such tables, allowing the TP to support up to 252 upstream ISPs.

The TP can connect to an upstream ISP over a point-to-point link using a variety of physical media or tunneling technologies. We also intend to support deployment of Transit Portals at exchange points, where the TP may connect with multiple ISPs over a local area network via a single interface. Each ISP in such shared media setup sends layer-two frames using a distinct source MAC address; the TP can use this address to correctly identify the sending ISP. Figure 6 shows how such traffic de-multiplexing is configured using policy routing rules. The ISP router has an IP address `1.1.1.1` with a MAC address `0:0:0:0:0:11` and a dedicated forwarding ta-

Requirement	Decision	Implementation
Transparent Connectivity (Section 3.1)		
Different layer-two connections	Tunnels between TP and virtual router	Tunneling technologies supported by the Linux kernel
Transparent traffic forwarding	Isolated forwarding tables for ISPs	Virtual forwarding tables and policy routing in Linux
Scalability (Section 3.2)		
Scalable routing with the # of ISPs	Isolated routing tables for ISPs	BGP views feature in Quagga `bgpd` daemon
Scalable updates with # of clients	Shared route update computation	Peer-group feature in Quagga `bgpd` daemon
Scalable forwarding with # of ISPs	Policy/default routing	Modifications to the Quagga `bgpd` daemon
Protection (Section 3.3)		
Preventing IP address spoofing	Packet filtering on source IP address	Linux `iptables`
Preventing prefix hijacking	Route filtering on IP prefix	Quagga prefix filters
Limiting routing instability	Rate-limiting of BGP update messages	Route-flap damping in Quagga `bgpd` daemon
Controlling bandwidth usage	Traffic shaping on virtual interfaces	Linux `tc`

Table 1: Design and implementation decisions.

ble, `1`. Line 1 shows the TP learning the MAC address of an upstream ISP when a new session is established. Then, lines 3–4 establish a policy-routing rule that redirects all the packets with this MAC address to a virtual forwarding table serving a new upstream ISP.

Transparency is another important goal for connectivity between client networks and the Transit Portal. In other words, a client network's connection to the TP should appear as though it were directly connected to the respective upstream networks. In the control plane, achieving this goal involves (1) removing the appearance of an extra AS hop along the AS path; and (2) passing BGP updates between client networks and upstreams as quickly as possible. The first task is achieved with the `remove-private-as rewrite` configuration (line 10 in Figure 7(a)), and the second task is achieved by setting the advertisement interval to a low value (line 18 in Figure 7(a)).

The Transit Portal supports clients regardless of whether they have a public or private AS number. To ensure transparency for the clients with a public AS number, the TP forwards the updates from such clients unmodified. Updates from clients with private AS numbers require rewriting.

3.2 Scalability

The TP maintains many BGP sessions, stores and disseminates many BGP routes, and forwards packets between many pairs of clients and ISPs. Scaling to a large number of ISPs and clients is challenging because each upstream ISP announces routes for many prefixes (*i.e.*, 300,000 routes); each client may receive routes from many (and possibly all) of these ISPs; and each client selects and uses routes independently. We describe three design decisions that we used to scale routing and forwarding at the TP: BGP views, peer groups, and default routing.

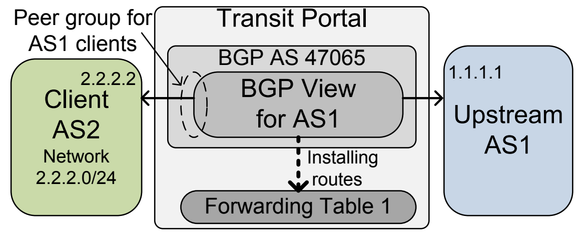
Scalable routing tables using BGP views. Rather than selecting a single best route for each destination prefix, the TP allows each service to select from the routes learned from all the upstream ISPs. This function requires the Transit Portal to disseminate routes from each ISP to the downstream clients, rather than selecting a single best route and could be achieved by having the TP run a separate instance of BGP for each upstream ISP, with BGP sessions with the ISP and each of the clients. Unfortunately, running multiple instances of BGP, each with its own process and associated state, would be expensive. Instead, the TP runs a single BGP instance with multiple "BGP views", each with its own routing table and decision process, for each upstream ISP. Using BGP views prevents the TP from comparing routes learned from different ISPs, while still capitalizing on opportunities to store redundant route information efficiently. Any downstream client that wants to receive routing messages from a specific upstream ISP need only establish a BGP session to the associated view. Our implementation uses the BGP `view` feature in Quagga; in particular, Figure 7(a) shows the configuration of a single "view" (starting in line 3) for upstream ISP 1. Section 5.2 shows that using BGP views in Quagga allows us to support approximately 30% more upstream ISPs with the same memory resources compared to the number of supported ISPs using conventional BGP processes.

Scalable updates to clients with BGP peer groups. Upon receiving a BGP update message from an upstream ISP, the TP must send an update message to each client that "connects" to that ISP. Rather than creating, storing, and sending that message separately for each client, the TP maintains a single BGP table and constructs a common message to send to all clients. Our implementation achieves this goal by using the `peer-group` feature in Quagga, as shown in Figure 7(a); in particular, line 14 associates `Client A` (CA) with the `peer-group` View1 for upstream ISP 1, as defined in lines 17–20. Note that although this example shows only one peer-group member, the real benefit of peer groups is achieved when multiple clients belong to the same group. Section 5.3 shows that peer-groups reduce CPU consumption threefold.

Smaller forwarding tables with default routes and policy routing. Despite storing and exchanging many BGP routes in the control plane, the Transit Portal should try to limit the amount of data-plane state for fast packet forwarding. To minimize data-plane state, the TP does *not* install all of the BGP routes from each BGP view in the kernel forwarding tables. Instead, the TP installs the smallest

```
 1  bgp multiple-instance
 2  !
 3  router bgp 47065 view Upstream1
 4    bgp router-id 47.0.0.65
 5    bgp forwarding-table 1
 6    bgp dampening
 7
 8    ! Connection to Upstream ISP
 9    neighbor 1.1.1.1 remote-as 1
10    neighbor 1.1.1.1 remove-private-AS rewrite
11    neighbor 1.1.1.1 attribute-unchanged as-path
          med
12
13    ! Connection to Downstream Client
14    neighbor 2.2.2.2 peer-group View1
15    neighbor 2.2.2.2 remote-as 2
16    neighbor 2.2.2.2 route-map CA-IN in
17    neighbor View1 peer-group
18    neighbor View1 advertisement-interval 2
19    neighbor View1 attribute-unchanged as-path med
20    neighbor View1 local-as 1 no-prepend
21  !
22  ip prefix-list CA seq 5 permit 2.2.2.0/24
23  !
24  route-map CA-IN permit 10
25    match ip address prefix-list CA
26  !
```

(a) Quagga configuration.



(b) Control-plane setup.

Figure 7: ***Example control-plane configuration and setup:*** **The TP is a hacked version of Quagga that installs non-default routes into Forwarding Table 1.**

amount of state necessary for custom packet forwarding to and from each client. On each virtual forwarding table, the TP stores only a default route to an upstream ISP associated with that table (to direct clients' outbound traffic through the ISP) and the BGP routes announced by the clients themselves (to direct inbound traffic from the ISP to the appropriate client). As shown in Section 5.2, this arrangement allows us to save about one gigabyte of memory for every 20 upstream ISPs. To selectively install only the routes learned from clients, rather than all routes in the BGP view, we make modifications to Quagga.

3.3 Protection

The TP must protect other networks on the Internet from misbehavior such as IP address spoofing, route hijacking or instability, or disproportionate bandwidth usage.

Preventing spoofing and hijacking with filters. The TP should prevent clients from sending IP packets or BGP route announcements for IP addresses they do not own. The TP performs ingress packet filtering on the source IP address and route filtering on the IP prefix, based on the client's address block(s). Our implementation filters packets using the standard `iptables` tool in Linux and filters routes using the `prefix-list` feature, as shown in lines 16 and 22-26 of Figure 7(a). In addition to filtering prefixes the clients do not own, the TP also prevents clients from announcing smaller subnets (*i.e.*, smaller than a /24) of their address blocks. Smaller subnets are also filtered by default by most of the Internet carriers. Section 7 describes how TP can overcome this limitation.

Limiting routing instability with route-flap damping. The TP should also protect the upstream ISPs and the Internet as a whole from unstable or poorly managed clients. These clients may frequently reset their BGP sessions with the TP, or repeatedly announce and withdraw their IP prefixes. The TP uses route-flap damping to prevent such instability from affecting other networks. Our implementation enables route-flap damping (as shown in line 6 of Figure 7(a)) with the following parameters: a half-life of 15 minutes, a 500-point penalty, a 750-point reuse threshold, and a maximum damping time of 60 minutes. These settings allow client to send the original update, followed by an extra withdrawal and an update, which will incur penalty of 500 points. Additional withdrawals or updates in a short timeframe will increase the penalty above reuse threshold and the route will be suppressed until the penalty shrinks to 750 points (the penalty halves every 15 minutes). There is no danger that one client's flapping will affect other clients, as route damping on the Internet operates separately for each announced route.

Controlling bandwidth usage with rate-limiting. The TP should prevent clients from consuming excessive bandwidth, to ensure that all clients have sufficient resources to exchange traffic with each upstream ISP. The TP prevents bandwidth hoarding by imposing rate limits on the traffic on each client connection. In our implementation, we use the standard `tc` (traffic control) features in Linux to impose a maximum bit rate on each client link.

4. Deployment

We have deployed Transit Portals in five locations. Four TPs are deployed in the United States, in Atlanta, Madison, Princeton, and Seattle. We also have one Transit Portal deployment in Tokyo, Japan. All Transit Portals are deployed in universities and research labs, whose networks act as a sole ISP in each location. Each ISP also provides full transit for our prefix and AS number. We are actively expanding this deployment: We are engaging with operators at two European research institutions and with one commercial operator in the U.S. to deploy more Transit Portals, and we are planning to expand our Seattle installation to connect to more ISPs.

The TPs advertise BGP routes using origin AS 47065 and IP prefix 168.62.16.0/21. Clients currently use a private AS number that the TP translates to the public AS number, 47065, before forwarding an update. Clients can also obtain their own AS number, in which case the TP re-advertises the updates without modification.

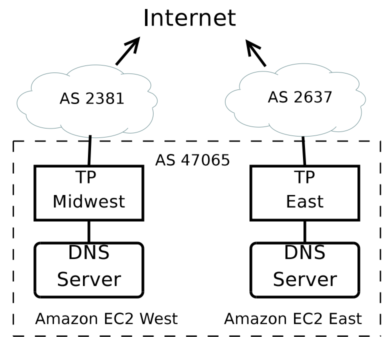


Figure 8: IP anycast experiment setup.

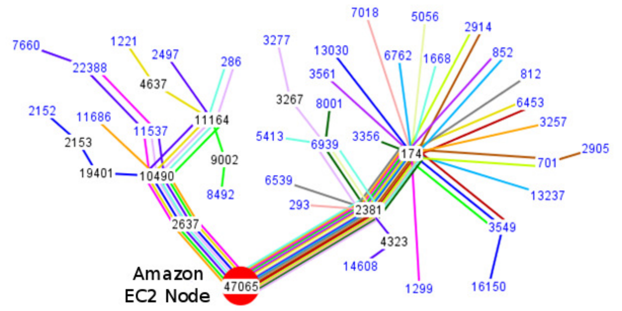


Figure 9: AS-level paths to an EC2 service sitting behind the Transit Portal (AS 47065), as seen in RouteViews.

This section presents the deployment of a distributed, anycasted DNS service, as we described in Section 2, that uses the TP for traffic control, similar to the service we described in Section 2 (Figure 2). In our evaluation of this deployment, we demonstrate two distinct functions: (1) the ability to load balance inbound and outbound traffic to and from the DNS service (including the ability to control the number of clients that communicate with each replica); and (2) the ability to reduce latency for specific subsets of clients with direct route control.

4.1 DNS With IP Anycast

In this section, we show how the TP delivers control and performance improvements for applications that can support IP anycast, such as anycast DNS resolution. The TP allows an IP anycast service to: (1) react to failures more quickly than using DNS re-mapping mechanisms, (2) load-balance inbound traffic, and (3) reduce the service response time. We explain the experiment setup and the measurements that show that adding IP anycast to services running on Amazon EC2 servers can improve latency, failover, and load-balance.

We deploy two DNS servers in Amazon EC2 data centers: one in the US-East region (Virginia) and another in the US-West region (Northern California). The servers are connected to two different TP sites and announce the same IP prefix to enable IP anycast routing as shown in Figure 8. The US-East region is connected to AS 2637 as an upstream provider, while the US-West region is connected to AS 2381 as its upstream provider. We measure the reachability and delay to these DNS servers by observing the response time to the IP anycast address from approximately 600 clients on different PlanetLab [21] nodes. Because our goal is to evaluate the scenario where the TP is collocated with a cloud computing facility, we adjust the measurements to discount the round-trip delay between the TP and the Amazon EC2 data centers.

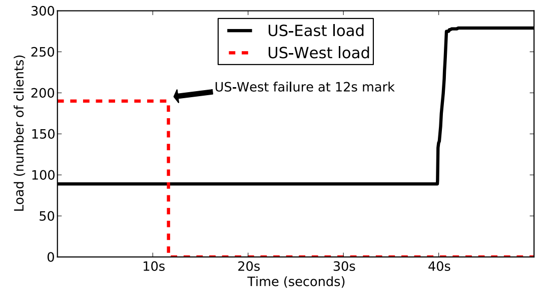


Figure 10: Failover behavior with two IP anycast servers.

The main provider of both upstreams is Cogent (AS 174), which by default prefers a downstream link to AS 2381. Cogent publishes a set of rules that allows Cogent's clients (*e.g.*, AS 2381, AS 2637, and their respective clients) to affect Cogent's routing choices [13]. The DNS service hosted on an Amazon host runs a virtual router and thus can apply these rules and control how incoming traffic ultimately reaches the service.

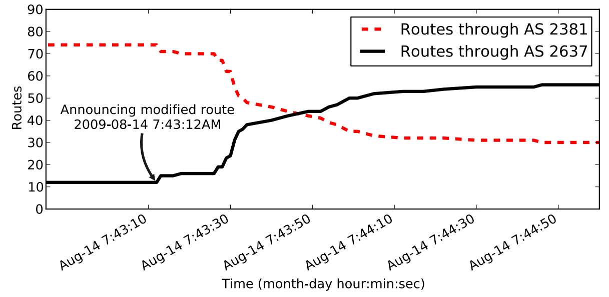
Figure 9 shows a capture from the BGPlay tool [1], which shows the initial routing state with the original BGP configuration. Most of the Internet paths to AS 47065 traverse Cogent (AS 174), which in turn prefers AS 2381 to forward the traffic to the client. Note that the client is configured with private AS number, but the TPs rewrite the updates before re-advertising them to the Internet. This rewriting causes the routers on the Internet to observe prefixes from as if they were announced by AS 47065.

Failover. Today's Internet applications use DNS name remapping to shift services to active data centers in the case of a data center or network failure. DNS name re-mapping is a relatively slow process because it requires the DNS entries to expire in DNS caches across the Internet. Applications that support IP anycast can rely on the routing infrastructure to route traffic to active data centers. In our experiment, we fail the server deployed in the US-West region and observe how quickly the clients converge to the US-East region.

Figure 10 shows how the load changes as we introduce failure. After twelve seconds, we fail the US-West deploy-

```
router bgp 65000
  neighbor 10.1.0.1 route-map OUT-2381 out
!
route-map OUT-2381 permit 10
  match ip address prefix-list OUR
  set community 174:10
```

(a) Route map.



(b) Route convergence after applying the route map.

Figure 11: Load balance: Applying a route map to outbound advertisements to affect incoming traffic.

	Avg. Delay	US-East	US-West
US-East	102.09ms	100%	0%
US-West	98.62ms	0%	100%
Anycast	94.68ms	42%	58%

Table 2: DNS anycast deployment. Average round trip time to the service and fraction of the load to each of the sites.

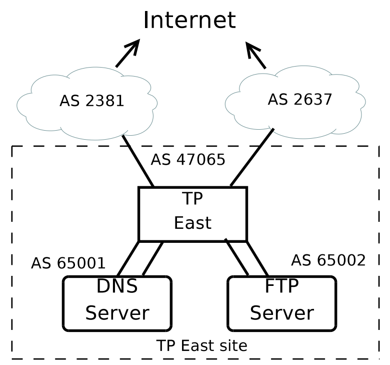


Figure 12: Outbound TE experiment setup.

ment and stop receiving requests at that site. After approximately 30 seconds, the routing infrastructure reroutes the traffic to the US-East site. The reaction to failure was automatic, requiring no monitoring or intervention from either the application or the TP operators.

Inbound traffic control. Assume that the DNS service would prefer most of its traffic to be served via AS 2637, rather than AS 2381. (The client network might prefer an alternate route as a result of cost, security, reliability, delay, or any other metric.) The Transit Portal clients can apply BGP communities to affect how upstream ISPs routes to its customers. On August 14, we changed the client configuration as shown in Figure 11(a) to add the BGP community `174:10` to a route, which indicates to one of the upstream providers, Cogent (AS 174), to prefer this route less than other routes to the client network.

To see how quickly the Internet converges to a new route, we analyze the route information provided by RouteViews. Figure 11(b) shows the convergence of the Internet routes to a new upstream. The dashed line shows the number of networks on the Internet that use the AS 2381 link, while the plain line shows the number of networks that use the AS 2637 link to reach the client hosted in the Amazon data center. (Note that the number of routes corresponds only to the routes that we collected from RouteViews.)

IP anycast application performance. We evaluate three DNS service scenarios: (1) US-East only, (2) US-West only, and (3) both servers using IP anycast routing. We measure the delay the PlanetLab clients observe to the IP anycast address in each of these scenarios. Using IP anycast should route each client to the closest active data center.

Table 2 shows the results of these experiments. Serving DNS using IP anycast improves response time by 4-8 milliseconds compared to serving from either of the sites separately. The improvement is not significant in our setup, since the Midwest and East Coast TP deployments are not far from each other. We expect more improvement when IP anycast is used from more diverse locations.

4.2 Outbound Traffic Control

We now show how two different services in a single data center can apply different outbound routing policies to choose different exit paths from the data center. Figure 12 shows the demonstration setup. DNS and FTP services run virtual routers configured as AS 65001 and AS 65002 respectively; both services and the Transit Portal are hosted at the same site as the ISP with an AS 2637. The ISP with an AS 2381 is in a different location and, for the sake of this experiment, the TP routes connections to it via a tunnel.

Without a special configuration, the services would choose the upstream ISP based on the shortest AS path. In our setup, we use the `route-map` command combined with a `set local-preference` setting to configure AS 65001 (DNS service) to prefer AS 2637 as an upstream and AS 65002 (FTP service) to prefer AS 2381 as an upstream. Figure 13 shows how the traceroutes to a remote host differ because of this configuration. The first hop in AS 65001 is a local service provider and is less than one millisecond away. The AS 65002 tunnels are transparently switched through a local TP and terminated at the remote AS 2381, which introduces additional delay.

```
AS65001-node:~# traceroute -n -q 1 -A
    133.69.37.5
traceroute to 133.69.37.5
 1  10.0.0.1 [*]  0 ms
 2  143.215.254.25 [AS2637]  0 ms
[skipped]
 8  203.181.248.110 [AS7660]  187 ms
 9  133.69.37.5 [AS7660]  182 ms
```

(a) Traceroute from AS 65001 client.

```
AS65002-node:~# traceroute -n -q 1 -A
    133.69.37.5
traceroute to 133.69.37.5
 1  10.1.0.1 [*]  23 ms
 2  216.56.60.169 [AS2381] 23 ms
[skipped]
 9  192.203.116.146 [*]  200 ms
 10  133.69.37.5 [AS7660]  205 ms
```

(b) Traceroute from AS 65002 client.

Figure 13: Traceroute from services co-located with TP East and AS 2637.

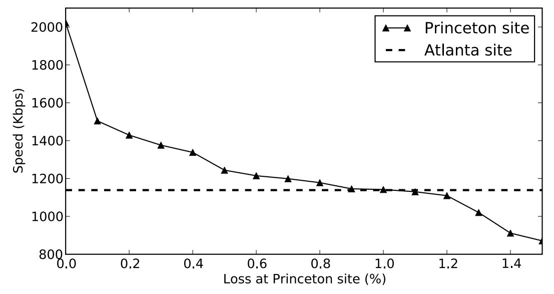


Figure 14: Average access speed from U.S. North East as packet loss is introduced at Princeton site.

4.3 Performance Optimization

The TP can be used to optimize the Internet service access performance. We simulate a video content provider with video streaming services running in cloud sites at the Princeton and the Atlanta locations. Bandwidth measurements show that the Princeton site offers better speed for clients in the northeast U.S., while the Atlanta site is preferred for the southeast U.S.

Assume that, due to periodic congestion, Princeton experiences intermittent packet loss every day around noon. Because the packet loss is intermittent, the application operator may be reluctant to use DNS to re-map the clients. Instead of DNS, the operator can use TP for rerouting when packet loss is detected. Figure 14 shows the average service access speed from the clients in the northeast as the loss at the Princeton site is increasing. As Princeton reaches a 1.1% packet loss rate, the Atlanta site, with its baseline speed of 1140 Kbps, becomes a better choice. Application operators then can use the methods described in Section 4.1 and 4.2 to reroute their applications when they observe losses in Princeton higher than 1.1%.

AS	Prefixes	Updates	Withdrawals
RCCN (1930)	291,996	267,207	15,917
Tinet (3257)	289,077	205,541	22,809
RIPE NCC (3333)	293,059	16,036,246	7,067
Global Crossing (3549)	288,096	883,290	68,185
APNIC NCC (4608)	298,508	589,234	9,147
APNIC NCC (4777)	294,387	127,240	12,233
NIX.CZ (6881)	290,480	150,304	11,247
AT&T (7018)	288,640	1,116,576	904,051
Hutchison (9304)	296,070	300,606	21,551
IP Group (16186)	288,384	273,410	47,776

Table 3: RIPE BGP data set for September 1, 2009.

5. Scalability Evaluation

This section performs micro-benchmarks to evaluate how the Transit Portal scales with the number of upstream ISPs and client networks. Our goal is to demonstrate the feasibility of our design by showing that a TP that is implemented in commodity hardware can support a large number of upstream ISPs and downstream clients. Our evaluation quantifies the number of upstream and client sessions that the TP can support and shows how various design decisions from Section 3 help improve scalability. We first describe our evaluation setup; we then explore how the number of upstream ISPs and downstream client networks affect the TP's performance for realistic routing workload. Our findings are unsurprising but comforting: A single TP node can support tens of upstream ISPs and hundreds of client networks using today's commodity hardware, and we do not observe any nonlinear scaling phenomena.

5.1 Setup

Data. To perform repeatable experiments, we constructed a BGP update dataset, which we used for all of our scenarios. We used BGP route information provided by RIPE Route Information Service (RIS) [23]. RIPE RIS provides two types of BGP update data: 1) BGP table dumps, and 2) BGP update traces. Each BGP *table dump* represents a full BGP route table snapshot. A BGP *update trace* represents a time-stamped arrival of BGP updates from BGP neighbors. We combine the dumps with the updates: Each combined trace starts with the stream of the updates that fill in the BGP routing table to reflect a BGP dump. The trace continues with the time-stamped updates as recorded by the BGP update trace. When we replayed this trace, we honored the inter-arrival intervals of the update trace.

Table 3 shows our dataset, which has BGP updates from 10 ISPs. The initial BGP table dump is taken on September 1, 2009. The updates are recorded in 24-hour period starting on midnight September 2 and ending at midnight September 3 (UTC). The average BGP table size is 291,869.1 prefixes. The average number of updates during a 24-hour period is 1,894,474.3, and the average number of withdrawals is 111,998.3. There are more announcements than withdrawals (a withdrawal occurs only if there is no alternate route to the prefix).

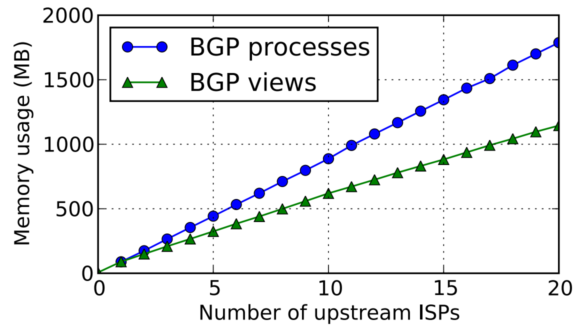


Figure 15: Control plane memory use.

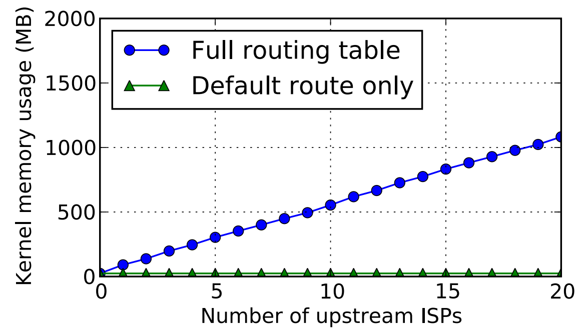


Figure 16: Data plane memory use.

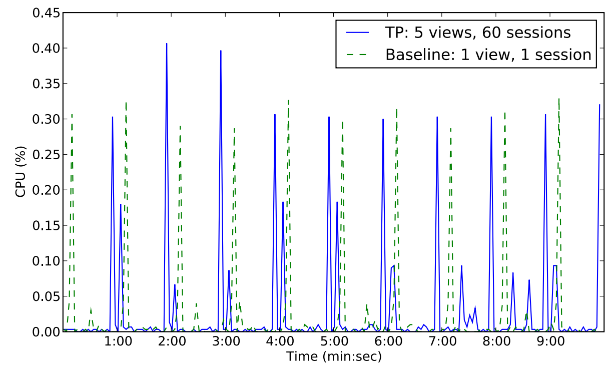


Figure 17: CPU usage over time (average taken every 3 seconds).

The data set contains two upstream ISPs with an unusually high number of updates: AS 3333 with more than 16 million updates, and AS 7018 with more than 900,000 withdrawals. It is likely that AS 3333 or its clients use reactive BGP load-balancing. In AS 7018, the likely explanation for a large number of withdrawals is a misconfiguration, or a flapping link. In any case, these outliers can stress the Transit Portal against extreme load conditions.

Test environment. We replayed the BGP updates using the `bgp_simple` BGP player [11]. The `bgp_simple` player is implemented using Perl `Net::BGP` libraries. We modified the player to honor the time intervals between the updates.

Unless otherwise noted, the test setup consists of five nodes: Two nodes for emulating clients, two nodes for emulating upstream ISPs, and one node under test, running the Transit Portal. The test node has two 1 Gbps Ethernet interfaces, which are connected to two LANs: one LAN hosts client-emulating nodes, the other LAN hosts upstream-emulating nodes. Each node runs on a Dell PowerEdge 2850, with a 3 Ghz dual-core 64-bit Xeon CPU and 2 GB of RAM. The machines run Fedora 8 Linux.

When we measured CPU usage for a specific process, we used the statistics provided by `/proc`. Each process has a `jiffies` counter, which records the number of system ticks the process used so far. For each test, we collect jiffies at five-second intervals over three minutes and the compute average CPU usage in percent. The `vmstat` utility provides the overall memory and CPU usage.

5.2 Memory Usage

Upstream sessions. Using a commodity platform with 2 GB of memory, TP scales to a few dozen of upstream ISPs. Figure 15 shows how the memory increases as we add more upstream ISPs. When TP utilizes separate BGP processes, each upstream ISP utilizes approximately 90MB of memory; using BGP views each upstream ISP utilizes approximately 60MB of memory. Data plane memory usage, as shown in Figure 16, is insignificant when using our forwarding optimization.

Downstream sessions. Each session to a downstream application consumes approximately 10MB of memory. For example, given 20 upstream ISPs, a client "subscribing" to all of them will consume 200MB. Upgrading the TP machine to 16GB of memory would easily support 20 upstream ISPs with more than a hundred clients subscribing to an average of 10 ISPs. The clients use only a small amount of memory in the data plane. The TP ensures forwarding only to the prefixes clients own or lease.

5.3 CPU Usage and Propagation Time

The main users of TP CPU are a BGP scan process, which scans the routes for the changes in reachability information, and BGP update parsing process, which parses the updates which arrive intermittently at a rate of approximately 2 million updates per day.

Figure 17 shows the timeseries of the CPU usage of two BGP processes as they perform routine tasks. Routing updates for both processes arrive from five different ISPs according to their traces. The baseline uses a default Quagga configuration with one routing table, and one client. The Transit Portal configuration terminates each ISP at a different virtual routing table and connects 12 clients (which amounts to a total of 60 client sessions). The TP configuration, on average consumes 20% more CPU than a baseline configuration; most of this load overhead comes from

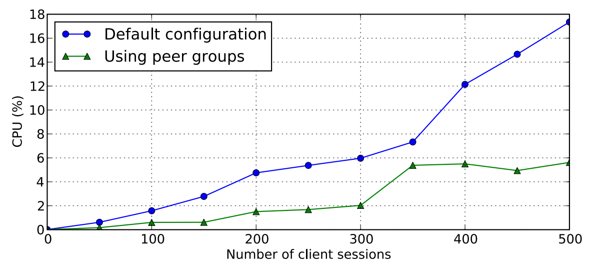


Figure 18: CPU usage while adding client sessions.

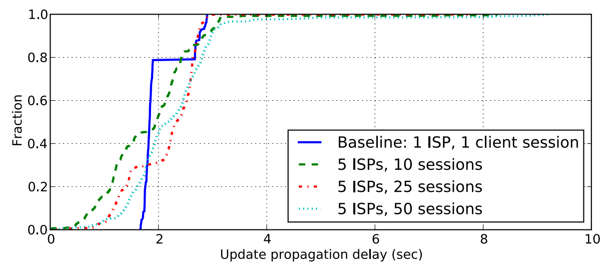


Figure 19: Update propagation delay.

maintaining more client sessions. The spikes in both plots correspond to a BGP scan that occurs every 60 seconds.

The TP can easily support hundreds of client BGP sessions. Figure 18 shows CPU usage as more client sessions are added. Two plots shows CPU usage using the default client configuration and CPU usage using a client configuration with a peer-group feature enabled. While conducting this experiment, we add 50 client sessions at a time and measure CPU load. We observe fluctuations in CPU use since at each measurement the number of updates passing the TP is slightly different. Nevertheless the trend is visible and each one hundred of client sessions increase CPU utilization by approximately a half of a percent.

Figure 19 shows the update propagation delays though the TP. The baseline configuration uses minimal configuration of Quagga with advertisement interval set to 2 seconds. Other configurations reflect the setup of five upstream providers with 10, 25, and 50 sessions. Approximately 40% of updates in the setup with 10 sessions are delivered within 1.6 seconds, while the baseline configuration seems to start deliver updates only at around 1.7 seconds due to the grouping of updates at the TP. A single upstream ISP sends updates in batches and each batch is subject to the configured two-second advertisement interval. When multiple upstream ISPs are configured, more updates arrive at the middle of advertisement interval and can be delivered as soon as it expires.

6. Framework for Provisioning Resources

The TP service provides an interface to the clients of existing hosting facilities to provision wide-area connectivity. In this section, we describe the design and implementation of this management plane. We first discuss the

```
1   <rspec type="advertisement" >
2
3    <node virtual_id="tp1">
4      <node_type type_name="tp">
5        <field key="encapsulation" value="gre"/>
6        <field key="encapsulation" value="egre"/>
7        <field key="upstream_as" value="1"/>
8        <field key="upstream_as" value="2"/>
9        <field key="prefix" count="3" length="24"/>
10     </node_type>
11   </node>
```

Figure 20: *Resource advertisement:* In Step 0 of resource allocation (Figure 21), the TP's component manager advertises available resources. This example advertisement says that the TP supports both GRE and EGRE encapsulation, has connections to two upstream ASes, and has three /24 prefixes available to allocate.

available resources and how they are specified. Next, we describe the process for clients to discover and request resources. Then, we discuss how we have implemented the management plane in the context of the GENI control framework [16]. In the future, the management plane will also control the hosting resources, and provide clients a single interface for resource provisioning.

6.1 Resources and Their Specification

The current resource allocation framework tracks *numbered* and *network* resources. The numbered resources include the available IP address space, the IP prefixes assigned to each client network, the AS number (or numbers) that each client is using to connect to the TPs, and which IP prefix will be advertised from each location. The framework must also keep track of whether a client network has its own IP prefix or AS number. Network resources include the underlying physical bandwidth for connecting to clients, and bandwidth available to and from each upstream ISP. Management of hosting resources, at this stage, is left for the client networks to handle.

The available resources should be specified in a consistent, machine-readable format. Our implementation represents resources using XML. Figure 20 shows an example advertisement, which denotes the resources available at one TP that offers two upstream connections, as indicated by lines 7–8 in the advertisement. The advertisement also indicates that this TP has three prefixes available for clients (line 9) and can support both GRE and EGRE tunneling (lines 5–6).

6.2 Discovering and Requesting Resources

Each Transit Portal runs a *component manager* (CM) that tracks available resources on the node. To track available capacity between TPs, or on links between virtual hosting facilities, the service uses an *aggregate manager* (AM). The aggregate manager maintains inventory over global resources by aggregating the available resources reported by the component managers. It also

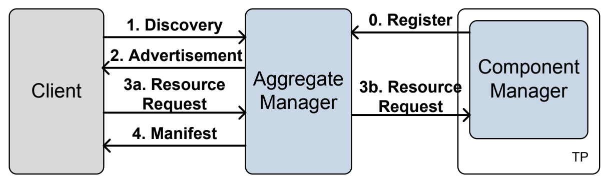


Figure 21: Resource allocation process.

brokers client requests by contacting individual CMs, as shown in Figure 21.

Clients can discover and request resources using a supplied command line tool `en-client.py`. The tool can issue resource discovery and resource reservation requests to a hard-coded AM address as shown in Section 4.2.

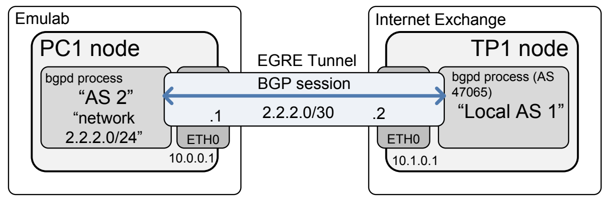
Before clients can request resources, the AM must know about resources in all TP installations. Each component manager *registers* with the AM and provides the list of the available resources, such as the list of upstream ISPs and available IP prefixes (Step 0 in Figure 21). To request resources, a client first issues a *discovery* call to an AM (Step 1). The AM replies with *advertisement*, which describes resources available for reservation (Step 1), such as the example in Figure 20. After receiving the resource advertisement, a client can issue a *resource request* (Step 3), such as the example in Figure 22(b). If the resources are available, the AM issues the *reservation request* to TP1 (Step 4) and responds with a *manifest* (Step 5), which is annotated version of the *request* providing the missing information necessary to establish the requested topology, as shown in Figure 22(c). The AM also provides sample client configuration excerpts with the manifest to streamline client configuration setup. The client uses the manifest to configure its end of the links and sessions, such as the configuration of PC1 in Figure 22(a).

6.3 Implementation

We implement the provisioning framework in the spirit of the Slice-based Facility Architecture (SFA) [5]. This management plane approach is actively developed by projects in the GENI [16] initiative, such as ProtoGENI [22]. SFA is a natural choice because our intermediate goal is to integrate the TP with testbeds like ProtoGENI [22] and VINI [25]. We use the *Twisted* event-driven engine libraries written in *Python* to implement the management plane components.

The primary components of the SFA are the Component Manager (CM) and Aggregate Manager (AM) as introduced before. The interface between the AM and CM is implemented using XML-RPC calls. The client interacts with the AM though a front-end, such as the Emulab or PlanetLab Web site, which in turn contacts the AM using XML-RPC or through the supplied client script.

Currently, access control to the AM is controlled with static access rules that authenticate clients and authorize



(a) Topology resulting from the resource request.

```
<rspec type="request" >
 <node virtual_id="tp1">
   <node_type type_name="tp">
     <field key="upstream_as" value="1" />
     <field key="prefix" count="1">
   </node_type>
 </node>
 <link virtual_id="link0">
   <link_type name="egre">
     <field key="ttl" value="255">
   </link_type>
   <interface_ref virtual_node_id="tp1" />
   <interface_ref virtual_node_id="pc1"
        tunnel_endpoint="10.0.0.1" />
 </link>
</rspec>
```

(b) The *resource request* specifies the client's tunnel endpoint, `10.0.0.1`, and asks for an EGRE tunnel, as well as an IP prefix and upstream connectivity to AS 1.

```
<rspec type="manifest" >
  <node virtual_id="tp1">
    <node_type type_name="tp">
      <field key="upstream_as" value="1" />
      <field key="prefix" count="1" value="
          2.2.2.0/24" />
    </node_type>
  </node>
  <link virtual_id="link0">
    <link_type name="egre">
      <field key="ttl" value="255" />
    </link_type>
    <interface_ref virtual_node_id="tp1"\
      tunnel_endpoint="10.1.0.1"\
      tunnel_ip="2.2.2.2/30" />
    <interface_ref virtual_node_id="pc1"\
      tunnel_endpoint="10.0.0.1"\
      tunnel_ip="2.2.2.1/30" />
  </link>
</rspec>
```

(c) The *manifest* assigns an IP prefix to the network, `2.2.2.0/24`, and specifies the parameters for the tunnel between PC1 and TP1.

Figure 22: The *resource request* (Step 3) and *manifest* (Step 5) of the resource allocation process, for an example topology.

or prevent a client from instantiating resources. To support more dynamic access control, we plan to expand the AM and CM to support security credentials, which will enable us to inter-operate with existing facilities (*e.g.*, PlanetLab, VINI, GENI) without using static access rules. We also plan to extend the resource management to include slice-based resource allocation and accounting.

7. Extensions to the Transit Portal

The TP is extensible. In the future, we plan to add support for lightweight clients who don't want to run BGP, support for smaller IP prefixes, support for backhaul between different TP sites, extensions for better scalability using hardware platforms for the data plane, and extensions for better routing stability in the face of transient client networks.

Support for lightweight clients. Some client networks primarily need to control traffic but do not necessarily need to run BGP between their own networks and the transit portal. In these cases, a client could use the existence or absence of a tunnel to the TP to signal to the TP whether it wanted traffic to enter over a particular ingress point. When the client network brings up a tunnel, the TP could announce the prefix over the appropriate ISP. When the client brings the tunnel down, the TP withdraws the prefix. As long as the tunnels are up, the client is free to choose an outgoing tunnel to route its traffic.

Support for small IP prefixes. Many client networks may not need IP addresses for more than a few hosts; unfortunately, these client networks would not be able to advertise their own IP prefix on the network, as ISPs typically filter IP prefixes that are longer than a /24 (*i.e.*, subnetworks with less than 256 addresses). The TP could allow client networks with fewer hosts to have BGP-like route control without having to advertise a complete /24 network. Clients for such networks would have full control of outgoing route selection and limited control for influencing incoming routes.

Better scalability. The scalability of the TP data plane can be further improved in two ways: (1) running multiple TPs in an Internet exchange, each serving subset of upstream ISPs, and (2) running the data and control plane of a TP on separate platforms. The first approach is easier to implement. The second approach offers the convenience of a single IP address for BGP sessions from ISPs and follows the best practices of data plane and control plane separation in modern routers. A data plane running on a separate platform could be implemented using OpenFlow or NetFPGA technologies.

Better routing stability in the face of transient client networks. The Transit Portal can support transient client networks that need BGP-based route control but do not need to use network resources all of the time. For example, suppose that a client network is instantiated every day for three hours to facilitate a video conference, or bulk transfer for backups. In these cases, the TP can simply leave the BGP prefix advertised to the global network, even when the client network is "swapped out". In this way, TPs could support transient client networks without introducing global routing instability.

Backhaul between sites. Today's cloud applications in different data centers, performing tasks such as backup or synchronization, must traverse the public Internet. For instance, Amazon EC2 platform offers sites in U.S. East coast, U.S. West coast and in Ireland. Unfortunately, the platform offers little transparency or flexibility for appli-

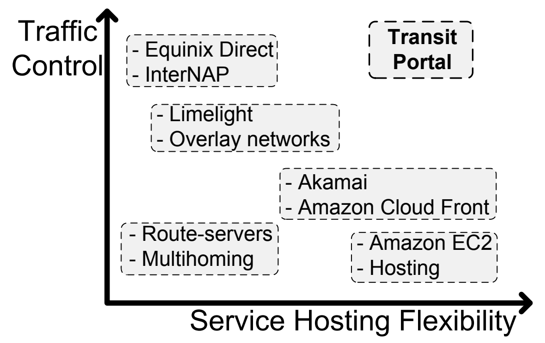


Figure 23: Transit Portals allow cloud providers to offer wide-area route control to hosted services

cation operators seeking to connect the applications in different sites into a common network. TP platform, on the other hand, is well suited to support sophisticated backhaul between applications in different sites. Each TP site can choose among multiple paths to other TP sites and application operator could exercise control on what path applications are routed to reach other sites. In addition, TP could facilitate private networking between the applications in different sites by using tunnels between TPs. The TP could also improve connectivity between the sites through overlay routing, two TP sites exchange traffic through a third intermediate TP site.

8. Related Work

The Transit Portal resembles several existing technologies, including content distribution networks, route servers, cloud hosting providers, and even exchange point operators. We describe how these existing technologies differ from TP, with respect to their support for the applications from Section 2.

Content distribution networks and cloud hosting providers do not provide control over inbound and outbound traffic. Content distribution networks like Akamai [6] and Amazon Cloud Front host content across a network of caching proxies, in an attempt to place content close to users to improve performance and save bandwidth. Each of these caching proxies may be located in some colocation facility with its own upstream connectivity. Some content providers may care more about throughput, others may care about low delay, others may care about reliability, and still others might care about minimizing costs. In a CDN, however, the content provider has no control over how traffic enters or leaves these colocation facilities; it is essentially at the mercy of the decisions that the CDN provider makes about upstream connectivity. The Transit Portal, on the other hand, allows each content provider to control traffic independently.

Exchange points do not provide flexible hosting. Providers like Equinix Direct [15] allow services to change their upstream connectivity on short timescales and connect on demand with ISPs in an Internet exchange. Equinix Direct operates only at the control plane and expects clients and ISPs to be able to share a common LAN. Unlike Equinix Direct, the Transit Portal allows services to establish connectivity to transit ISPs without renting rack space in the exchange point, acquiring numbered resources, or procuring dedicated routing equipment.

Route servers do not allow each downstream client network to make independent routing decisions. Route servers reduce the number of sessions between the peers in an exchange point: instead of maintaining a clique of connections, peers connect to a central route server. Route servers aggregate the routes and select only the best route to a destination to each of the peers [18]. This function differs from the TP, which provides transparent access to all of the routes from upstream ISPs.

DNS-based load balancing cannot migrate live connections. Hosting providers sometimes use DNS-based load balancing to redirect clients to different servers—for example, a content distribution network (*e.g.*, Akamai [6]) or service host (*e.g.*, Google) can use DNS to re-map clients to machines hosting the same service but which have a different IP address. DNS-based load balancing, however, does not allow the service provider to migrate a long-running connection, and it requires the service provider to use low DNS TTLs, which may also introduce longer lookup times. The Transit Portal, on the other hand, can move a service by re-routing the IP prefix or IP address associated with that service, thereby allowing for longer DNS TTLs or connection migration.

Overlay networks do not allow direct control over inbound or outbound traffic, and may not scale. Some control over traffic might be possible with an overlay network (*e.g.*, RON [9], SOSR [19]). Unfortunately, overlay networks can only indirectly control traffic, and they require traffic to be sent through overlay nodes, which can result in longer paths.

9. Conclusion

This paper has presented the design, implementation, evaluation, and deployment of the Transit Portal, which offers flexible wide-area route control to for distributed services. The TP prototype is operating at several geographically diverse locations with a /21 address block, AS number, and BGP sessions to upstream providers. We have used Transit Portal in both our research and in projects for a graduate course [**?**], and we plan to deploy and evaluate additional services, including offering our platform to other researchers, and to offer new, more lightweight interfaces to the Transit Portal.

Acknowledgments

We thank Alex Fabrikant, Steven Ko, Ellen Zegura, and our shepherd, Andrew Warfield, for their feedback. This work was funded by NSF award CNS-0626950, NSF CAREER award 0694974, and a grant from the GENI project office "Bringing Experimenters and External Connectivity to GENI".

REFERENCES

[1] BGPlay Route Views. `http://bgplay.routeviews.org/bgplay/`.
[2] Gaikai Demo. `http://www.dperry.com/archives/news/dp_blog/gaikai_-_video/`, 2010.
[3] ooVoo. `http://www.oovoo.com/`, 2010.
[4] Skype. `http://www.skype.com/`, 2010.
[5] Slice-based facility architecture. `http://www.cs.princeton.edu/~llp/arch_abridged.pdf`, 2010.
[6] Akamai. http://www.akamai.com, 2010.
[7] A. Akella, B. Maggs, S. Seshan, A. Shaikh, and R. Sitaraman. A measurement-based analysis of multihoming. In *Proc. ACM SIGCOMM*, Karlsruhe, Germany, Aug. 2003.
[8] A. Akella, J. Pang, B. Maggs, S. Seshan, and A. Shaikh. A comparison of overlay routing and multihoming route control. In *Proc. ACM SIGCOMM*, Portland, OR, Aug. 2004.
[9] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Banff, Canada, Oct. 2001.
[10] M. Armbrust, A. Fox, R. Grifth, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California at Berkeley, Feb. 2009.
[11] bgpsimple: simple BGP peering and route injection script . `http://code.google.com/p/bgpsimple/`.
[12] Cisco Optimized Edge Routing (OER). `http://www.cisco.com/en/US/products/ps6628/products_ios_protocol_option_home.html`, 2010.
[13] Cogent Communications BGP Communities. `http://www.onesc.net/communities/as174/`, 2010.
[14] Emulab. `http://www.emulab.net/`.
[15] Equinix Direct. `http://www.equinix.com/solutions/connectivity/equinixdirect/`, 2010.
[16] GENI: Global Environment for Network Innovations. `http://www.geni.net/`.
[17] D. K. Goldenberg, L. Qiu, H. Xie, Y. R. Yang, and Y. Zhang. Optimizing cost and performance for multihoming. In *Proc. ACM SIGCOMM*, pages 79–92, Portland, OR, Aug. 2004.
[18] R. Govindan, C. Alaettinoglu, K. Varadhan, and D. Estrin. Route Servers for Inter-Domain Routing. *Computer Networks and ISDN Systems*, 30:1157–1174, 1998.
[19] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.
[20] U. of Oregon. RouteViews. `http://www.routeviews.org/`.
[21] PlanetLab. `http://www.planet-lab.org/`, 2010.
[22] ProtoGENI. `http://www.protogeni.net/`, 2010.
[23] Réseaux IP Européens Next Section Routing Information Service (RIS). `http://www.ripe.net/ris/`.
[24] A. Schran, J. Rexford, and M. Freedman. Namecast: A Reliable, Flexible, Scalable DNS Hosting System. *Princeton University, Technical Report TR-850-09*, 2009.
[25] VINI: Virtual Network Infrastructure. `http://www.vini-veritas.net/`.
[26] A. Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007.

LiteGreen: Saving Energy in Networked Desktops Using Virtualization

Tathagata Das
tathadas@microsoft.com
Microsoft Research India

Pradeep Padala[*]
ppadala@docomolabs-usa.com
DOCOMO USA Labs

Venkata N. Padmanabhan
padmanab@microsoft.com
Microsoft Research India

Ramachandran Ramjee
ramjee@microsoft.com
Microsoft Research India

Kang G. Shin
kgshin@eecs.umich.edu
The University of Michigan

Abstract

To reduce energy wastage by idle desktop computers in enterprise environments, the typical approach is to put a computer to sleep during long idle periods (e.g., overnight), with a proxy employed to reduce user disruption by maintaining the computer's network presence at some minimal level. However, the Achilles' heel of the proxy-based approach is the inherent trade-off between the functionality of maintaining network presence and the complexity of application-specific customization.

We present *LiteGreen*, a system to save desktop energy by virtualizing the user's desktop computing environment as a virtual machine (VM) and then migrating it between the user's physical desktop machine and a VM server, depending on whether the desktop computing environment is being actively used or is idle. Thus, the user's desktop environment is "always on", maintaining its network presence fully even when the user's physical desktop machine is switched off and thereby saving energy. This seamless operation allows LiteGreen to save energy during short idle periods as well (e.g., coffee breaks), which is shown to be significant according to our analysis of over 65,000 hours of data gathered from 120 desktop machines. We have prototyped LiteGreen on the Microsoft Hyper-V hypervisor. Our findings from a small-scale deployment comprising over 3200 user-hours of the system as well as from laboratory experiments and simulation analysis are very promising, with energy savings of 72-74% with LiteGreen compared to 32% with existing Windows and manual power management.

1 Introduction

The energy consumed by the burgeoning computing infrastructure worldwide has recently drawn significant attention. While the focus of energy management has been on the data-center setting [20, 29, 32], attention has also been directed recently to the significant amounts of energy consumed by desktop computers in homes and enterprises [17, 31]. A recent U.S. study [33] estimates that PCs and their monitors consume about 100 TWh/year, constituting 3% of the annual electricity consumed in the U.S. Of this, 65 TWh/year is consumed by PCs in enterprises, which constitutes 5% of the commercial building electricity consumption in the U.S. Moreover, market projections suggest that PCs will continue to be the dominant desktop computing platform, with over 125 million units shipping each year from 2009 through 2013 [15].

The usual approach to reducing PC energy wastage is to put computers to sleep when they are idle. However, the presence of the user makes this particularly challenging in a desktop computing environment. Users care about preserving long-running network connections (e.g., login sessions, IM presence, file sharing), background computation (e.g., syncing and automatic filing of new emails), and keeping their machine reachable even while it is idle. Putting a desktop PC to sleep is likely to cause disruption (e.g., broken connections), thereby having a negative impact on the user, who might then choose to disable the energy savings mechanism altogether.

To reduce user disruption while still allowing machines to sleep, one approach has been to have a *proxy* on the network for a machine that is asleep [33]. However, this approach suffers from an inherent tradeoff between functionality and complexity because of the need for application-specific customization.

In this paper, we present *LiteGreen*, a system to save desktop energy by employing a novel approach to minimizing user disruption and avoiding the complexity of application-specific customization. The basic idea is to virtualize the user's desktop computing environment, by encapsulating it in a virtual machine (VM), and then migrating it between the user's physical desktop machine and a VM server, depending on whether the desktop computing environment is actively used or idle. When the desktop becomes idle, say when the user steps away for several minutes (e.g., for a coffee break), the desktop VM is migrated to the VM server and the physical desktop machine is put to sleep. When the desktop becomes active again (e.g., when the user returns), the desktop VM is migrated back to the physical desktop machine. Thus, even when it has been migrated to the VM server, the user's desktop environment remains alive (i.e., it is "always on"), so ongoing network connections and other activity (e.g., background downloads) are *not* disturbed, regardless of the application involved.

[*]The author was an intern at MSR India during the course of this work.

The "always on" feature of LiteGreen allows energy savings whenever the opportunity arises, without having to worry about disrupting the user. Besides long idle periods (e.g., nights and weekends), energy can also be saved by putting the physical desktop computer to sleep even during short idle periods, such as when a user goes to a meeting or steps out for coffee. Indeed, our measurements indicate that the potential energy savings from exploiting short idle periods are significant (Section 3).

While the virtualization-based approach allows keeping the desktop environment "always on", two key challenges need to be addressed for it to be useful for saving energy on desktop computers. First, how do we provide a normal (undisrupted) desktop experience to users, masking the effect of VMs and their migration? Second, how do we decide when and which VMs to migrate to/from the server in order to maximize energy savings while minimizing disruption to users?

To address the first challenge, LiteGreen uses the *live migration* feature supported by modern hypervisors [21] coupled with the idea of always presenting the desktop environment through a level of indirection (Section 4). Thus, whether the VM is at the server or desktop, users always access their desktop VM through a remote desktop (RD) session. So, in a typical scenario, when a user returns to their machine that has been put to sleep, the machine is woken up from sleep and the user is able to immediately access their desktop environment (whose state is fully up-to-date, because it has been "always on") through an RD connection to the desktop VM running on the VM server. Subsequently, the desktop VM is migrated back to the user's physical desktop machine without the user even noticing.

To address the second challenge, LiteGreen uses an energy-saving algorithm that runs on the server and carefully balances migrations based on two continuously-updated lists: 1) VMs in the *mandatory to push* list must be migrated to the desktop machine to minimize user disruption, and 2) VMs in the *eligible to pull* list may be migrated to server for energy savings, subject to server capacity constraints (Section 5).

We have prototyped LiteGreen on the Microsoft Hyper-V hypervisor (Section 6). We have a small-scale deployment running on the desktop machines of ten users, comprising three administrative staff and seven researchers, including three authors of this paper. A demonstration video of LiteGreen is available at [4]. Separately, we have conducted laboratory experiments using both the Hyper-V and Xen hypervisors to evaluate various aspects of LiteGreen. We have also developed a simulator to analyze the data we gathered and to understand the finer aspects of our algorithms.

We have analyzed (a) over 65,000 user-hours of data gathered by us from 120 desktop computers at Microsoft Research India (MSRI), and (b) 3200 user-hours of data from a deployment of our prototype on ten user desktops over a span of 28 days. Based on this analysis, LiteGreen is able to put desktop machines to sleep for 86-88% of the time, resulting in an estimated energy savings of 72-74%. In comparison, through a combination of manual user action and the automatic Windows power management, desktop machines are put to sleep for 35% of time, delivering estimated energy savings of only 32%.

The main contributions of this paper are as follows:

1. A novel system that leverages virtualization to consolidate idle desktops on a VM server, thereby saving energy, while avoiding user disruption.
2. Automated mechanisms to drive the migration of the desktop computing environment between the physical desktop machines and the VM server.
3. A prototype implementation and the evaluation of LiteGreen through a small-scale deployment on the desktops of ten users, spaning 3200 user-hours over 28 days, yielding energy savings of 74%.
4. Trace-driven analysis of over 65,000 user-hours of resource usage data gathered from 120 desktops, yielding energy savings of 72%, with short idle periods ($<$ 3 hours) contributing 20% or more.

2 Problem Background and Related Work

In this section, we provide some background on the problem setting and discuss related work.

2.1 PC Energy Consumption

Researchers have measured and characterized the energy consumed by desktop computers [17]. The typical desktop PC consumes 80-110 W when active and 60-80 W when idle, excluding the monitor, which adds another 35-80 W. The relatively small difference between active and idle modes is significant and arises because the processor itself only accounts for a small portion of the total energy. In view of this, multiple S ("sleep") states have been defined as part of the ACPI standard [13]. In particular, the S3 state ("standby") suspends the machine's state to RAM, thereby cutting energy consumption to 2-3 W. S3 has the advantage of being much quicker to transition in and out of than S4 ("hibernate"), which involves suspending the machine's state to disk.

2.2 Proxy-based Approach

As discussed above, the only way of cutting down the energy consumed by a PC is to put it to sleep. However, when a PC it put to sleep, it loses its network presence, resulting in disruption of ongoing connections (e.g., remote-login or file-download sessions) and the machine even becoming inaccessible over the network.

The resulting disruption has been recognized as a key reason why users are often reluctant to put their machines to sleep [17]. Researchers have found that roughly 60% of office desktop PCs are left on continuously [33].

The general approach to allowing a PC to sleep while maintaining some network presence is to have a network proxy operate on its behalf while it is asleep [33]. The functionality of the proxy could span a wide range:

- **WoL Proxy:** The simplest proxy allows the machine to be woken up using the *Wake-on-LAN* (WoL) mechanism [12] supported by most Ethernet NICs. To be able to send the "magic" WoL packet, the proxy must be on the same subnet as the target machine and needs to know the MAC address of the machine. Typically, machine wakeup is initiated manually.
- **Protocol Proxy:** A more sophisticated proxy performs automatic wakeup, triggered by a filtered subset of the incoming traffic [31, 34]. The filters could be configured based on user input and also the list of network ports that the target machine was listening on before it went to sleep. Other traffic is either responded to by the proxy itself without waking up the target machine (e.g., ARP for the target machine) or ignored (e.g., ARP for other hosts).
- **Application Proxy:** An even more sophisticated proxy incorporates application-specific stubs that allow it to engage in network communication on behalf of applications running on the machine that is now asleep [31]. Such a proxy could even be integrated into an augmented NIC [17].

Enhanced functionality of a proxy comes at the cost of greater complexity, for instance, the need to create stubs for each application that the user wishes to keep alive. LiteGreen sidesteps this complexity by keeping the entire desktop computing environment alive, by consolidating it on the server along with other idle desktops. On the flip side, however, LiteGreen is more heavyweight than the proxy approach, as we discuss in Section 9.2.

2.3 Saving Energy through Consolidation

Consolidation to save energy has been employed in other computing settings—data centers and thin clients.

In the data-center setting, server consolidation is used to approximate energy proportionality by migrating computation, typically using virtualization, from several lightly-loaded servers onto fewer servers, and then turning off the servers that are freed up [20, 37, 38]. Doing so saves not only the energy consumed directly by the servers but also the significant amount of energy consumed indirectly for cooling [29, 30].

Thin client based computing, an idea that is making a reappearance [23, 11] despite failures in the past, represents an extreme form of consolidation, with all of the computing resources being centralized. While the cost, management, and energy savings might make the model attractive in some environments, there remain questions regarding the up-front hardware investment needed to migrate to thin clients. Also, thin clients represent a trade-off and may not be suitable in settings where power users want the flexibility of a PC or insulation from even transient dips in performance due to consolidation. Indeed, market projections suggest that PCs will continue to be the dominant desktop computing platform, with over 125 million units shipping each year from 2009 through 2013 [15], and with thin clients replacing only 15% of PCs by 2014 [14]. Thus, there will continue to be a sizeable and growing installed base of PCs for the foreseeable future, possibly as part of mixed environments comprising both PCs and thin clients, so addressing the problem of energy consumed by desktop PCs remains important.

While LiteGreen's use of consolidation is inspired by the above work, a key difference arises from the presence of users in a desktop computing environment. Unlike in a data center setting, where machines tend to run server workloads and hence are substitutable to a large extent, a desktop machine is a user's *personal* computer. Users expect to have access to *their* computing environment. Furthermore, unlike in a thin client setting, users expect to have good interactive performance and the flexibility of attaching specialized hardware and peripherals (e.g., a high-end graphics card). Progress on virtualizing high-end hardware, such as GPUs [24, 28], facilitates LiteGreen's approach of running the desktop in a VM.

Central to the design of LiteGreen is preserving this PC model and minimizing both user disruption and new hardware cost, by only consolidating idle desktops.

2.4 Virtualization and Live Migration

A key enabler of consolidation is virtualization. Several hypervisors are available commercially [2, 5, 8]. These leverage the hardware support that modern processors include for virtualization [3, 1].

Virtualization has simplified the task of moving computation from one physical machine to another [40] compared to process migration [36]. Efficient live migration over a high-speed LAN is performed by iteratively copying memory pages while the VM continues execution, before finally pausing the VM briefly (for as short as 60 ms [21]) to copy the remaining pages and resume execution on the destination machine. Live migration has been extended to wide-area networks as well [27].

2.5 Page Sharing and Memory Ballooning

Consolidation of multiple VMs on the same physical server can put pressure on the server's memory resources. *Page sharing* is a technique to decrease the memory footprint of VMs by sharing pages that are in common across multiple VMs [39]. Recent work [26] has advanced the state of the art to also include sub-page level sharing, yielding memory savings of up to 90% with homogeneous VMs and up to 65% otherwise.

Even with page sharing, memory can become a bottleneck depending on the number of VMs that are consolidated on the server. *Memory ballooning* is a technique to dynamically shrink or grow the memory available to a VM with minimal overhead relative to statically provisioning the VM with the desired amount of memory [39].

2.6 Virtualization in LiteGreen Prototype

For our LiteGreen prototype, we use the Microsoft Hyper-V hypervisor. While this is a server hypervisor, the ten users in our deployment were able to use it without difficulty for desktop computing. Since Hyper-V currently does not support page sharing or memory ballooning, we conducted a separate set of experiments with the Xen hypervisor to evaluate memory ballooning. Finally, since Hyper-V only supports live migration with shared storage, we set up a shared storage server connected to the same GigE switch as the desktop machines and the server (see Section 9 for a discussion of shared storage).

3 Motivation Based on Measurement

To provide concrete motivation for our work beyond the prior work discussed above, we conducted a measurement study on the usage of PCs. Our study was set in the MSR India lab during the summer of 2009, at which time the lab's population peaked at around 130 users. Of these, 120 users at the peak volunteered to run our measurement tool, which gathered information on the PC resource usage (in terms of the CPU, network, disk, and memory) and also monitored user interaction (UI). In view of the sensitivity involved in monitoring keyboard activity on the volunteers' machines, we only monitored mouse activity to detect UI.

We have collected over 65,000 hours worth of data from these users. We placed the data gathered from each machine into 1-minute buckets, each of which was then annotated with the level of resource usage and whether there was UI activity. We classify a machine as being *idle* (as opposed to being *active*) during a 1-minute bucket using one of the two policies discussed later in Section 5.2: the *default* policy, which only looks for the absence of UI activity in the last 10 minutes, and a more *conservative* policy, which additionally checks whether the CPU usage was below 10%.

Based on this data, we seek to answer the following questions:

Q1. How (under)utilized are desktop PCs?

To help answer this question, Figure 1a plots the distribution of CPU usage and UI activity, binned into 1-minute buckets and aggregated across all of the PCs in our study. To allow plotting both CPU usage and UI activity in the same graph, we adopt the convention of treating the presence of UI activity in a bucket as 100% CPU usage. The "CPU only" curve in the figure shows that CPU usage is low, remaining under 10% for 90% of the time. The "CPU + UI" curve shows that UI activity is present, on average, only in 10% of the 1-minute buckets, or about 2.4 hours in a day. However, since even an active user might have 1-minute buckets with no UI activity (e.g., they might just be reading from the screen), the total UI activity is very likely larger than 10%.[1]

While both CPU usage and UI activity are low, it still does not mean that the PC can be simply put to sleep, as we discuss below.

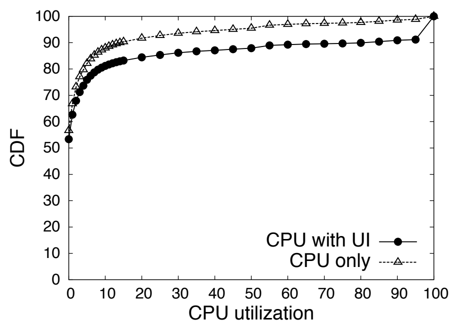
Q2. How are the idle periods distributed?

Given that there is much idleness in PCs, the next question is how the idle periods are distributed. We define an idle period as a contiguous sequence of 1-minute buckets, each of which is classified as being idle. The conventional wisdom is that idle periods are long, e.g., overnight periods and weekends. Figure 1c shows the distribution of idle periods based on the default (UI only) and conservative (UI and CPU usage) definitions of idleness noted above. Each data point shows the aggregate idle time (shown on the y axis on a log scale) spent in idle periods of the corresponding length (shown on the x axis). The x axis extends to 72 hours, or 3 days, which encompasses idle periods stretching over an entire weekend.

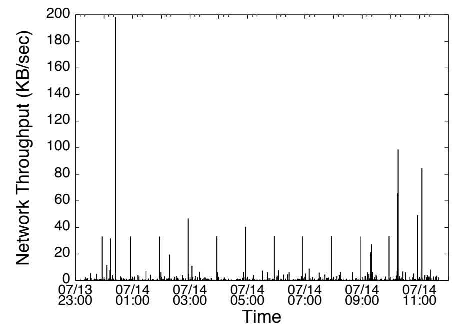
The default curve shows distinctive peaks at around 15 hours (overnight periods) and 63 hours (weekends). It also shows a peak for short idle periods, under about 3 hours in length. In the conservative curve, the peak at the short idle periods dominates by far. The overnight and weekend peaks are no longer distinctive since, based on the conservative definition of idleness, these long periods tend to be interrupted, and hence broken up, by intervening bursts of background CPU activity.

Figure 1d shows that with the default definition of idleness, idle periods shorter than 3 hours add up to about 20% of the total duration of idle periods longer than 3 hours. With the conservative policy, the short idle

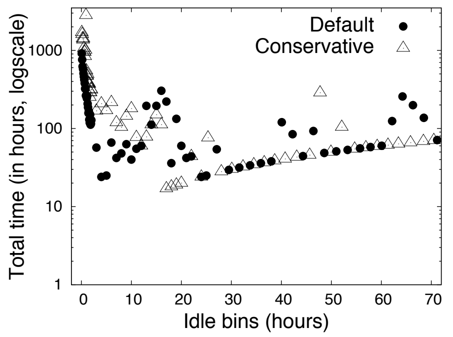
[1] It is possible that we may have missed periods when there was keyboard activity but no mouse activity. However, we ran a test with a small set of 3 volunteers, for whom we monitored keyboard activity as well as mouse activity, and found it rare to have instances, where there was keyboard activity but no mouse activity in the following 10 minutes.



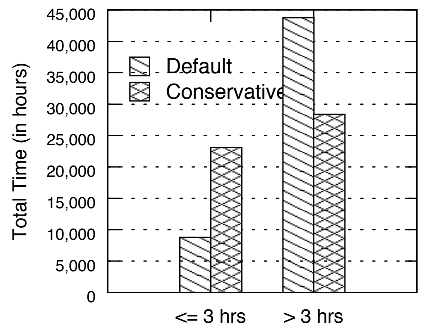
(a) Distribution of CPU utilization



(b) Network activity during the night on one idle desktop machine



(c) Distribution of idle periods



(d) Comparison of aggregate duration of short and long idle periods

Figure 1: Analysis of PC usage data at MSR India

Category	Example Applications	Sleep	Proxy-based On-demand Wakeup	LiteGreen
Incoming requests	incoming RDP	fails	works but with initial delay/timeout	works
	file share			works but requires disk
Idle connections	outgoing RDP	broken connection		works
	IM	user shown as offline		user shown as away
Background tasks	large file download	download stalled ⇒ delay		works
	software patching (e.g., Windows update)	patch download delay ⇒ larger window of vulnerability		patches downloaded but need disk for application

Table 1: Impact of various energy saving strategies on applications

periods add up to over 80% of the total duration of the long idle periods. Thus, the short idle periods, which correspond to lunch breaks, meetings, etc., during a work day, represent a significant opportunity for energy savings over and above the savings from the long idle periods considered in prior work.

Q3. Why not just sleep during idle periods?

Even when the machine is mostly idle (i.e., has low CPU utilization), it could be engaged in network activity, as depicted in Figure 1b. A closer look at this machine (with the owner's permission) revealed that the processes that showed sporadic activity were (a) `InoRT.exe`, a virus scanner, (b) `DfrgNtfs.exe`, a disk defragmenter, (c) `TrustedInstaller.exe`, which checks for Windows software updates, and (d) `Svchost.exe`, which encapsulates miscellaneous services. Putting the machine to sleep would delay or disrupt these tasks, possibly incovenencing the user.

Privacy considerations prevented us, in general, from gathering detailed information such as process names, which would have revealed the identities of the applications running on a user's machine. Hence, we use indirect means to understand how sleep might be disruptive.

Through informal conversations at MSR India, we compiled a list of typical applications that users run. Table 1 categorizes these and reports on the impact of sleep on these applications. We find that the applications suffer disruption to varying degrees. In some cases, sleep causes a hard failure, e.g., a broken connection. In other cases, it causes a soft failure. For example, if a user steps out for a meeting and their (idle) machine goes to sleep, IM might show them, somewhat misleadingly, as being "offline" when "away" would be more appropriate.

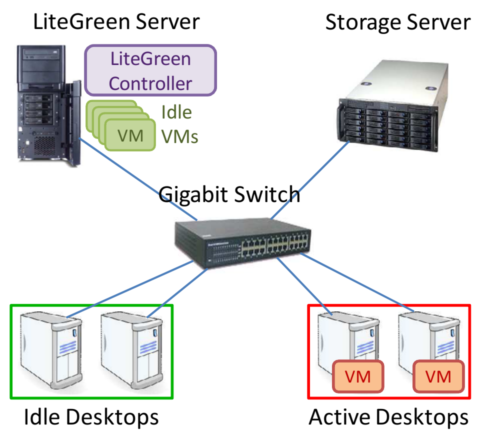


Figure 2: **LiteGreen architecture**: Desktops are in active (switched on) or idle (sleep) state. Server hosts idle desktops running in VMs

The ability to do on-demand wakeup, as provided by a proxy, helps when there is new inbound communication, e.g., an incoming remote desktop (RDP) connection. Such communication would work, although it might suffer from an initial delay or timeout owing to the time it takes to wake up from sleep. However, with applications where there is an ongoing connection, the proxy approach is unable to prevent disruption. In fact, the only way of avoiding disruption is to not go to sleep, which means giving up on energy savings.

Avoiding disruption requires that the applications continue to run and maintain their network presence even while the machine is (mostly) idle. Doing so while still saving energy motivates a solution such as LiteGreen. In some cases, however, LiteGreen would require access to the local disk, either immediately (e.g., file sharing) or eventually (e.g., software patching). While our current implementation does not migrate the disk, we believe that such migration is feasible, as discussed in Section 9.

In summary, we make two key observations from our analysis. First, desktop PCs are often idle, and there is significant opportunity to exploit short idle periods. Second, it is important to maintain network presence even during the idle periods to avoid user disruption.

4 System Architecture

Figure 2 shows the high-level architecture of LiteGreen. The desktop computing infrastructure is augmented with a VM server and a shared storage node. In general, there could be more than one VM server and/or shared storage node. All of these elements are connected via a high-speed LAN such as Gigabit Ethernet.

Each desktop machine as well as the server run a hypervisor. The hypervisor on the desktop machine hosts a VM in which the client OS runs. This VM is migrated to the server when the user is not active and the desktop is put to sleep. When the user returns, the desktop is woken up and the VM is "live migrated" back to the desktop. To insulate the user from such migrations, the desktop hypervisor also runs a remote desktop (RD) client [7], which is used by the user to connect to, and remain connected to, their VM, regardless of where it is running. Although our current prototype does not leverage it, the advent of GPU virtualization [24, 28] allows improving the user experience by bypassing remote desktop when the VM is running locally on the desktop machine.

The user's desktop VM uses, in lieu of a local disk, the shared storage node, which is also shared with the server. This aspect of the architecture arises from the limitations of live migration in hypervisors currently in production and can be done away with once live migration with local VHDs is supported (Section 9).

The hypervisor on the server hosts the guest VMs that have been migrated to it from (idle) desktop machines. The server also includes a *controller*, which is the brain of LiteGreen. The controller receives periodic updates from *stubs* on the desktop hypervisors on the level of user and computing activity on the desktops. The controller also tracks resource usage on the server. Using all of this information, the controller orchestrates the migration of VMs to the server and back to the desktop machines, and manages the allocation of resources on the server. We have chosen a centralized design for the controller because it is simple, efficient, and also enables optimal migration decisions to be made based on full knowledge (e.g., the bin-packing optimization noted in Section 5.3).

5 Design

Having provided an overview of the architecture, we now detail the design of LiteGreen. The design of LiteGreen has to deal with two somewhat conflicting goals: maximizing energy savings from putting machines to sleep while minimizing disruption to users. When faced with a choice, LiteGreen errs on the side of being conservative, i.e., avoiding user disruption even if it means reduced energy savings.

The operation of LiteGreen can be described in terms of a control loop effected by the controller based on local information at the server as well as information reported by the desktop stubs. We discuss the individual elements before putting together the whole control loop.

5.1 Which VMs to Migrate?

The controller maintains two lists of VMs:

- *Eligible for Pull:* list of (idle) VMs that currently reside on the desktop machines but could be migrated (i.e., "pulled") to the server, thereby saving energy without user disruption.
- *Mandatory to Push:* list of (now active) VMs that had previously been migrated to the server but must now be migrated (i.e., "pushed") back to the desktop machines at the earliest to minimize user disruption.

In general, the classification of a VM as active or idle is made based on both UI activity initiated by the user and computing activity, as discussed next.

5.2 Determining If Idle or Active

The presence of any UI activity initiated by the user, through the mouse or the keyboard (e.g., mouse movement, mouse clicks, key presses), in the recent past (*actvityWindow*, set to 10 minutes by default) is taken as an indicator that the machine is active. Even though the load imposed on the machine might be rather minimal, we make this conservative choice to reflect our emphasis on minimizing the impact on the interactive performance perceived by the user.

In the *default policy*, the presence of UI activity is taken as the *only* indicator of whether the machine is active. So, the absence of recent UI activity is taken as an indication that the machine is idle.

A more *conservative policy*, however, also considers the actual computational load on the machine. Specifically, if the CPU usage is above a threshold, the machine is deemed to be active. So, for the machine to be deemed idle, both the absence of recent UI activity and CPU usage being below the threshold are necessary conditions. To avoid too much bouncing between the active and idle states, we introduce hysteresis in the process by (a) measuring the CPU usage as the average over an interval (e.g., 1 minute) rather than instantaneously, and (b) having a higher threshold, c_{push}, for the push list (i.e., idle$\rightarrow$active transition of a VM currently on the server) than the threshold, c_{pull}, for the pull list (i.e., for a VM currently on a desktop machine).

5.3 Server Capacity Constraint

A second factor that the controller considers while making migration decisions is the availability of resources on the server. If the server's resources are saturated or close to saturation, the controller migrates some VMs back to the desktop machines to relieve the pressure. Thus, an idle VM is merely *eligible* for being consolidated on the server and, in fact, might not be if the server does not have the capacity. On the other hand, an active VM must be migrated back to the desktop machine even if the server has the capacity. This design reflects the choice to err on the side of being conservative, as noted above.

There are two server resource constraints that we focus on. The first is *memory availability*. Given a total server memory, M, and the allocation, m, made to each VM, the number of VMs that can be hosted on the server is bounded by $n_{RAM} = \frac{M}{m}$. Note that m is the memory allocated to a VM after ballooning and would typically be some minimal value such as 384 MB that allows an idle VM to still function (Section 7.4).

The second resource constraint arises from *CPU usage*. Basically, the aggregate CPU usage of all the VMs on the server should be below a threshold. As with the conservative client-side policy discussed in Section 5.2, we introduce hysteresis by (a) measuring the CPU usage as the average over a time interval (e.g., 1 minute), and (b) having a higher threshold, s_{push}, for pushing out VMs, than the threshold, s_{pull}, for pulling in VMs. The server tries to pull in VMs (assuming the pull list is non-empty) so long as the aggregate CPU usage is under s_{pull}. Then, if the CPU usage rises above s_{push}, the server pushes back VMs. Thus, there is a bound, n_{CPU}, on the number of VMs that can be accommodated such that $\sum_{i=1}^{i=n_{CPU}} x_i \leq s_{push}$, where x_i is the CPU usage of the i^{th} VM.

The total number of VMs that can be consolidated on the server is bounded by $min(n_{RAM}, n_{CPU})$. While one could extend this mechanism to other resources such as network and disk, our evaluation in Section 8 indicates that enforcing CPU constraints also ends up limiting the usage of other resources.

Instead of simply pulling in VMs until the capacity limit is reached, more sophisticated optimizations are possible. In general, the problem of consolidating VMs within the constraints of the server's resources can be viewed as a bin-packing problem [25] since consolidating the multiple new VMs in place of the one that is evicted would likely help save energy. Details of our greedy bin packing algorithm for managing consolidation are described in [22].

5.4 Measuring & Normalizing CPU Usage

Given the heterogeneity of desktop and server physical machines, one question is how CPU usage is measured and how it is normalized across the machines. All measurement of CPU usage in LiteGreen, both on the server and on the desktop machines, is made at the hypervisor level, where the controller and stubs run, rather than within the guest VMs. Besides leaving the VMs untouched and also accounting for CPU usage by the hypervisor itself, measurement at the hypervisor level has the advantage of being unaffected by the configuration of the virtual processors. The hypervisor also provides uniform interface to interact with multiple operating systems.

Another issue is normalizing measurements made on the desktop machines with respect to those made on the server. For instance, when a decision to pull a VM is made based on its CPU usage while running on the desktop machine, the question is what its CPU usage would be once it has been migrated to the server. In our current design, we only normalize at the level of cores, treating cores as equivalent regardless of the physical machine. So, for example, a CPU usage of $x\%$ on a 2-core desktop machine would translate to a CPU usage of $\frac{x}{4}\%$ on an 8-core server machine. One could consider refining this design by using the CPU benchmark numbers for each processor to perform normalization.

5.5 Putting It All Together: LiteGreen Control Loop

To summarize, LiteGreen's control loop operates as follows. Based on information gathered from the stubs, the controller determines which VMs, if any, have become idle, and adds them to the pull list. Furthermore, based both on information gathered from the stubs and from local monitoring on the server, the controller determines which VMs, if any, have become active again and adds these to the push list. If the push list is non-empty, the newly active VMs are migrated back to the desktop right away. If the pull list is non-empty and the server has the capacity, additional idle VMs are migrated to the server. If at any point, the server runs out of capacity, the controller looks for opportunities to push out the most expensive VMs in terms of CPU usage and pull in the least expensive VMs from the pull list. Pseudocode for the control loop employed by the LiteGreen controller is available at [22].

6 Implementation and Deployment

We have built a prototype of LiteGreen based on the Hyper-V hypervisor, which is available as part of the Microsoft Hyper-V Server 2008 R2 [5]. The Hyper-V server can host Windows, Linux, and other guest OSes and also supports live migration based on shared storage.

Our implementation comprises the controller, which runs on the server, and the stubs, which run on the individual desktop machines. The controller and stubs are written in C# and add up to 1600 and 600 lines of code, respectively. The stubs use WMI (Windows Management Instrumentation) [10] and Powershell to perform the monitoring and migration. The controller also includes a GUI, which shows the state of all of the VMs in the system.

In our implementation, we ran into a few issues from bugs in the BIOS to limitations of Hyper-V and had to work around them. Here we discuss a couple of these.

Lack of support for sleep in hypervisor: Since Hyper-V is intended for use on servers, it does not support sleep once the hypervisor service has been started. Also, once started, the hypervisor service cannot be turned off without a reboot. Other hypervisors such as Xen also lack support for sleep.

We worked around this as follows: when the desktop VM has been migrated to the server and the desktop machine is to be put to sleep, we set a registry key to disable the hypervisor and then reboot the machine. When the machine boots up again, the hypervisor is no longer running, so the desktop machine can be put to sleep. Later, when the user returns and the machine is woken up, the hypervisor service is restarted, without requiring a reboot. Since a reboot is needed only when the machine is put to sleep but *not* when it is woken up, the user does not perceive any delay or disruption due to the reboot.

BIOS bug: On one model of desktop (Dell Optiplex 755), we found that the latest version of BIOS available does not restore prior-enabled Intel VT-x support (needed by the hypervisor) after resuming from sleep. We are currently pursuing a fix to this issue with the manufacturer; until then, we are unable to use this model of desktop as a LiteGreen client.

6.1 Deployment

We have deployed LiteGreen to ten users at MSR India, comprising three administrative staff and seven researchers, three of whom are authors of this paper. As of this writing, the system has been in use for 28 days that includes 10 weekend days and holidays. Accounting for the ramp-up and ramp-down of users in the LiteGreen system, total usage was approximately 3200 user-hours.

Each user is given a separate LiteGreen desktop machine that is running a hypervisor (Hyper-V Server 2008) along with the LiteGreen client stub. The desktop environment runs in a Windows 7 VM that is allocated 2GB of memory. The users' existing desktop is left untouched in order to preserve the users' existing desktop configuration and local data. Different users use their LiteGreen desktop in different ways. Most users use the LiteGreen desktop as their primary access to computing, relying on remote desktop to connect to their existing desktop. A couple of users used it only for specific tasks, such as browsing or checking email, so that the LiteGreen desktop only sees a subset of their activity.

Our findings are reported in Section 7.3. While our deployment is very small in size and moreover, has not entirely replaced the users' existing desktop machines, we believe it is a valuable first step that we plan to build on in the coming months. A video clip of LiteGreen in action on one of the desktop machines is available at [4].

7 Experimental Evaluation

We begin by presenting experimental results based on our prototype. These results are drawn both from controlled experiments in the lab and from our deployment. The results are, however, limited by the small scale of our testbed and deployment, so in Section 8 we present a

Component	Make/Model	Hardware	Software
Desktops (10)	HP WS xw4600	Intel E8200 Core 2 Duo @2.66GHz	Hyper-V + Win7 guest
Server	HP Proliant ML350	Intel Xeon E5440 DualProc 4Core 2.83GHz, 32GB RAM	Hyper-V
Storage	Dell Optiplex 755	Intel E6850 Core 2 Duo 3.00 GHz	Win 2008 + iSCSI
Switch	DLink DGS-1016D	NA	NA

Table 2: Testbed details

larger scale trace-driven evaluation using the traces gathered from the 120 machines at our lab.

7.1 Testbed

Our testbed mirrors the architecture depicted in Figure 2. It comprises ten desktop machines, a server, and a storage node, all connected to a GigE switch. The hardware and software details are listed in Table 2.

We first used the testbed for controlled experiments in the lab (Section 7.2). We then used the same setup but with the desktop machines installed in the offices of the participating users, for our deployment (Section 7.3).

7.2 Results from Laboratory Experiments

We start by walking through a migration scenario similar to that shown in the LiteGreen video clip [4], before presenting detailed measurements.

7.2.1 Migration Timeline

The scenario, shown in Figure 3a, starts with the user stepping away from his/her machine (Event A), causing the machine to become idle. After *actvityWindow* amount of time elapses, the user's VM is marked as idle and the server initiates the VM pull (Event B). After the VM migration is complete (Event C), the physical desktop machine goes to sleep (Event D). Note that, if the user returns to their desktop between events B and C, the migration is simply canceled without any perceivable latency to the user. This is because, during live migration, the (desktop) VM continues to remain fully operational, except during the final switchover phase that typically lasts only tens of milliseconds.

Figure 3b shows the timeline for waking up. When the user returns to his/her desktop, the physical machine wakes up (Event A) and immediately establishes a remote desktop (RD) session to the user's VM (Event B). At this point, the user can start working even though his/her VM is still on the server. A VM push is initiated (Event C) and the VM is migrated back to the physical desktop machine (Event D), in the background using live migration feature.

Figures 3a and 3b also show the power consumed by the desktop machine and the server over time, measured using Wattsup power meters [9]. While the timeline shows the measurements from one run, we also made more detailed measurements of the individual components and operations, which we present next.

7.2.2 Timing of Individual Operations

We made measurements of the time taken for the individual steps involved in migration. In Table 3, we report the results derived from ten repetitions of each step.

Step	Sub-step	Time (sec) [mean (sd)]
Going to Sleep		840.5 (27)
	Pull Initiation	638.8 (20)
	Migration	68.5 (5)
	Sleep	133.2 (5)
Resuming from sleep		164.6 (16)
	Wakeup	5.5
	RD connection	14
	Push Initiation	85.1 (17)
	Migration	60 (6)

Table 3: Timing of individual steps in migration

7.2.3 Power Measurements

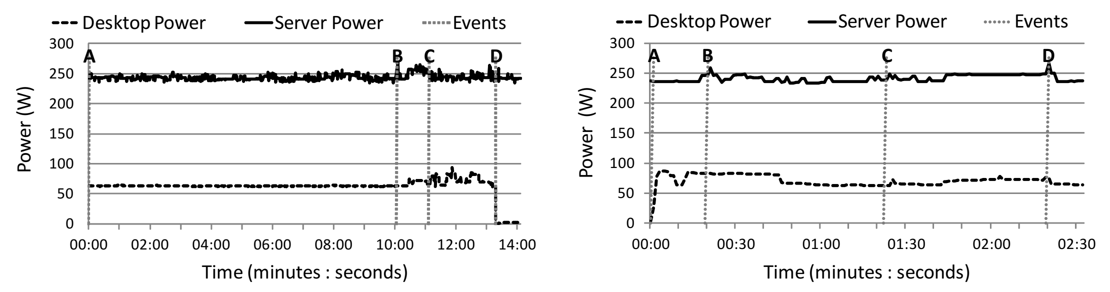
Table 4 shows the power consumption of a desktop machine, the server, and the switch in different modes, measured using a Wattsup power meter.

Component	Mode	Power (W)
Desktop	idle	60-65W
Desktop	100% CPU	95W
Desktop	sleep	2.3-2.5W
Server	idle	230-240W
Server	100% CPU	270W
Switch	idle	8.7 - 8.8W
Switch	during migration	8.7-8.8W

Table 4: Power measurements

The main observation is that power consumption of the desktop and the servers is largely unaffected by the amount of CPU load. It is only when the machine is put to sleep that the power drops significantly. We also see that the power consumption of the network switch is low and is unaffected by any active data transfers. Thus, the energy cost of the migration itself is negligible (the small bump between events B and C in Figure 3a), and can be ignored, as long as one accounts for the time/energy of the powered on desktop until the migration is completed.

Finally, the power consumption curves in Figures 3a and 3b show the marked difference in the impact of migration on the power consumed by the desktop machine



(a) Sleep timeline (b) Wakeup timeline

Figure 3: Migration timelines

and the server. When a pull happens, the power consumed by the desktop machine goes down from about 60W in idle mode to 2.5W in sleep mode (with a transient surge to 75W during the migration process). On the other hand, the power consumption of the server barely changes. This difference underlies the significant net energy gain to be had from moving idle desktops to the server.

7.2.4 Compression to Reduce Migration Time

The time to migrate the VM — either push or pull — is determined by the memory size (2GB) of the VM and the network throughput. The transfer size can be greater than memory size when application activity during the time of migration results in dirty memory pages that are copied multiple times. We configured a desktop VM with typical enterprise applications, such as Microsoft Office, an email client and a browser with multiple open web pages. We then migrated this VM back and forth, between the desktop and the server. When the VM was on the desktop, we interacted with the applications as a regular desktop user. In this setup, we observed that different migrations resulted in transfer sizes between 2.2-2.7GB. Using a network transfer rate of 0.5Gbps (the effective TCP throughput of active migration on the GigE network), transfer takes about 35-43 seconds. Including the migration initiation overhead, the total migration time is about 60 seconds, which matches the numbers shown in the timeline and in Table 3.

We experimented with a simple compression optimization to reduce the migration time. We used EndRE [16], an end-system redundancy elimination service, with a 250MB packet cache to analyze the savings from performing redundancy elimination in the VM migration traffic between two nodes. EndRE works in a similar fashion to WAN optimizers [18], but on end hosts instead of middleboxes. After identifying and eliminating redundant bytes, as small as 32 bytes, with respect to the packet cache, GZIP is applied to further compress the data. For various transfers, we found that the compressor, operating at 0.4Gbps, was able to reduce the size of transfer by 64-69%. Note that EndRE is designed to be asymmetric. Thus, decompression is inexpensive and does not result in additional latency. This implies that *migration transfer time can be reduced from 35-43 seconds to about 15 seconds using redundancy elimination, thereby significantly speeding up the migration process.* This approach can also help support migration of VMs with larger memory sizes (e.g., 4GB) while limiting the transfer time to under a minute.

7.2.5 Further Optimizations

First, the time to put the machine to sleep is 133 seconds, much of it due to the reboot necessitated by the lack of support for sleep in Hyper-V (Section 6). With a *client* hypervisor that includes support for sleep, we expect the time to go to sleep to shrink to just about 10 seconds.

Second, the time from when the user returns till when they are able to start working is longer than we would like — about 19.5 seconds. Of this, resuming the desktop machine from sleep only constitutes 5.5 seconds. About 4 more seconds are taken by the user to key in their login credentials. The remaining 10 seconds are taken to launch the remote desktop application and make a connection to the user's VM, which resides on the server. This longer than expected duration is because Hyper-V freezes for several seconds after resuming from sleep. We believe that this happens because our unconventional use of Hyper-V, specifically putting it to sleep when it is not designed to support sleep, triggers some untested code paths. We expect that this issue would be resolved with a client hypervisor. However, resuming the desktop and connecting to the users' VM may still take on the order of a few seconds that may be disruptive. One approach to mask this disruption is to anticipate user returns, for example, through user mobile phone tracking, and resume the desktop before the user arrives at his disk. This aspect is discussed in [22]. Such tracking of the user' location could also be used to improve user experience in other days, for instance, by preventing a seemingly idle machine from being migrated to the server, say, if the user is still in their office.

7.3 Results from Deployment

As noted in Section 6.1, we have had a deployment of LiteGreen for a period of 28 days including 10 holidays

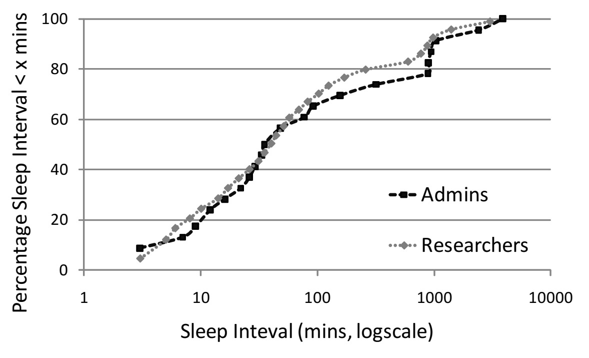


Figure 4: Distribution of desktop sleep durations

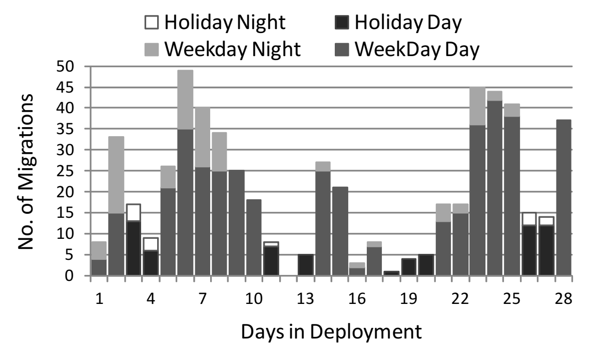


Figure 5: Number of migrations

and weekends, comprising about 3200 user-hours (maximum simultaneous usage by 10 users). While it is hard to draw general conclusions given the small scale and duration of the deployment thus far, it is nevertheless interesting to consider some of the initial results.

7.3.1 Desktop Sleep Time Distribution

Figure 4 shows the cumulative distribution function of the sleep durations for the seven researchers and the three administrative staff. The sleep times tend to be quite short, with a median of 40 minutes across the ten users, demonstrating the exploitation of short idle periods by the LiteGreen system. From the figure, one can see one distinct difference in behavior between the administrators and the researchers in our study. We notice that there is a sharper spike in the curve for the administrators around 900 minutes as compared to the smoother curve for researchers. This is explained by the fact that administrators are more likely than researchers to maintain regular workhours (e.g., 9AM to 6PM) which corresponds to 15 hours (900 minutes) of idle time.

7.3.2 Desktop Average Sleep Time

For our deployment, we used the default policy from Section 5.2 to determine whether a VM was idle or active. During the deployment period, the desktop machines were able to sleep for an average of 87.9% of the time. Even the machine of the most active user in our deployment, who used their LiteGreen desktop for all of their computing activity, slept for 76% of the time.

Note that, while 88% of desktop sleep time may appear unusually large, out of the 3200 user-hours, only about 960 user-hours corresponded to daytime weekdays (8AM – 8PM) in our deployment. Thus, 12% or 384 user-hours of desktop awake time corresponds to 40% of daytime weekday hours, representing a significant fraction of the workday.

7.3.3 Energy Savings

The conversion of desktop average sleep time to energy savings requires accounting of the energy costs of the server. While a LiteGreen server was necessary for this deployment, it was significantly under-utilized since it was dedicated to host only 10 idle VMs. If we amortize the cost of the server over a larger number of desktops (e.g., 60), the power cost of the server per desktop is 4.2W (see Section 8.4 for details). We use this amortized value of the server power cost below.

We use the power measurement numbers from Table 4 to estimate energy savings from LiteGreen. Let us assume power consumption of 62.5W for an idle desktop, 95W for a fully active desktop and 2.5W for a sleeping desktop. From Figure 1a, where CPU usage is less than 10% for 90% of the time, let us assume a desktop that never sleeps consumes 62.5W of power 90% of the time and 95W of power 10% of the time. Then, desktop power consumption, without any energy savings, is simply (0.9*62.5+0.1*95) = 65.75W per desktop.

In LiteGreen, since the average desktop sleep time is 88%, the power savings is (0.88*(62.5 - 2.5) - 4.2) = 48.6 W per desktop or 74% of total desktop energy consumption.

Finally, the above energy savings calculations are applicable for enterprises that already have a centralized storage deployment. Otherwise, we need to take into account the energy consumed by the centralized storage system as well. Consider a network attached storage box such as the QNAP SS-839 Pro Turbo [6] that can host up to 8 disks and consumes 34W in operation. Assuming two desktop users are multiplexed onto each disk, each of these storage devices can support up to 16 desktops. Thus, the amortized energy cost of centralized storage is 34/16 = 2.1W/desktop. Accounting for the storage overhead, the power savings in LiteGreen is 48.6 - 2.1 = 46.5W per desktop or 71%.

7.3.4 Number of Migrations

Finally, Figure 5 shows the number of migrations for the different days of deployment, segregated by daytime (8 am–8 pm) and nighttime (8 pm–8 am), and further classified by weekdays and holidays (including weekends). There were a total of 571 migrations during the deployment period. The number of migrations are higher during

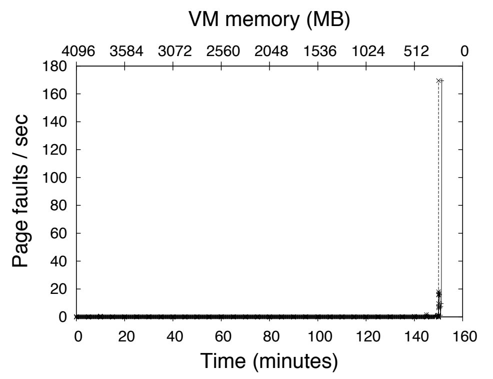


Figure 6: Memory ballooning experiment: every 5 minutes memory of a desktop VM is reduced by 128M. Initial memory size is 4096M

the day compared to night (470 versus 101) and higher in the weekdays compared to the holidays (493 versus 78). These numbers are again consistent with the LiteGreen approach of exploiting short idle intervals.

7.4 Experiments with Xen

We would like to evaluate the effectiveness of memory ballooning in relieving pressure on the server's memory resources due to consolidation. However, Hyper-V does not currently support memory ballooning, so we conducted experiments using the Xen hypervisor, which supports memory ballooning for the Linux guest OS using a balloon driver (we are not aware of any balloon driver for Windows). We used the Xen hypervisor (v3.4.2 built with 2.6.18 SMP kernel) with the Linux guest OS (CentOS 5.4) on a separate testbed comprising two HP C-class blades, each equipped with two quad-core 2.2 GHz 64-bit processors with 48GB memory, two Gigabit Ethernet cards, and two 146 GB disks. One blade was used as the desktop machine and the other as the server.

The desktop Linux VM was initially configured with 4096 MB of RAM. It ran an idle workload comprising the Gnome desktop environment, two separate Firefox browser windows, with a Gmail account and the CNN main page open (each of these windows auto-refreshed periodically without any user's involvement), and the user's home directly mounted through SMB (which also generated background network traffic). The desktop VM was migrated to the server. Then, memory ballooning was used to shrink the VM's memory all the way down to 128 MB, in steps of 128 MB every 5 minutes.

Figure 6 shows the impact of memory ballooning on the page fault rate. The page fault rate remains low even when the VM's memory is shrunk down to 384 MB. However, shrinking it down to 256 MB causes the page fault rate to spike, presumably because the working set no longer fits within memory.

We conclude that in our setup with the idle workload that we used, memory ballooning could be used to shrink the memory of an idle VM by over a factor of 10 (4096 MB down to 384 MB), without causing thrashing. Further savings in memory could be achieved through memory sharing. While we were not able to evaluate this in our testbed since neither Hyper-V nor Xen supports it, the findings from prior work [26] are encouraging, as discussed in Section 9.

8 Trace-driven Analysis

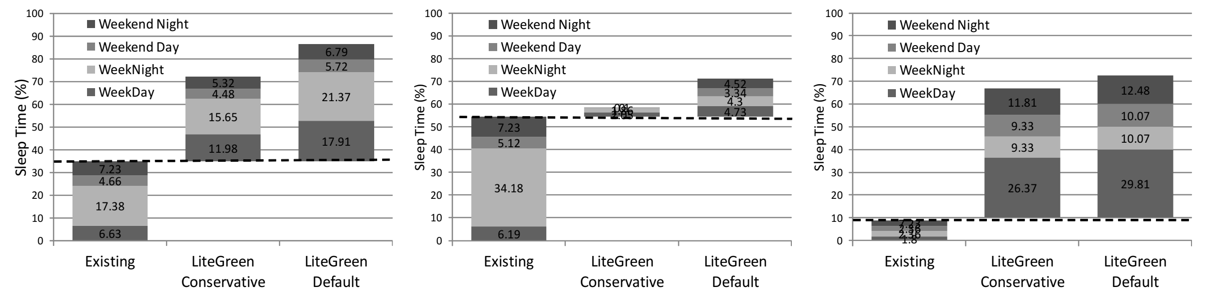
To evaluate our algorithms further, we have built a discrete event simulator written in Python using the SimPy package. The simulator runs through the desktop traces, and simulates the default and conservative policies based on various parameters including c_{pull} (client resource threshold below which client VMs are eligible to be pulled to server), c_{push} (client resource threshold above which client VMs are pushed to client) , s_{pull} (server resource thrshold above below client VMs are pulled to server), s_{push} (server resource threshold above which client VMs are pushed to clientdis) and $ServerCapacity$. In the rest of the section, we will report on energy savings achieved by LiteGreen and utilization of various resources (CPU, network, disk) at the server as a result of consolidation of the idle desktop VMs.

8.1 Desktop Sleep Time

Figure 7a shows the desktop sleep time for all the users with existing mechanisms and LiteGreen, default and conservative. For both the policies, we use $c_{pull} = 10$ (less than 10% desktop usage classified as idle), $c_{push} = 20$, $s_{pull} = 600$, $s_{push} = 700$ and $ServerCapacity = 800$ intended to represent a Server with 8 CPU cores.

As mentioned earlier, our desktop trace gathering tool records a number of parameters, including CPU, memory, UI activity, disk, network, etc., every minute after its installation. In order to estimate energy savings using existing mechanisms (either automatic windows power management or manual desktop sleep by the user), we attribute any unrecorded interval or "gaps" in our desktop trace to energy savings via existing mechanisms. Using this technique, we estimate that existing mechanisms would have put desktops to sleep 35.2% of the time.

We then simulate the migrations of desktop VMs to/from the server depending on the desktop trace events and the above mentioned thresholds for the conservative and default policy. Using the conservative policy, we find that LiteGreen puts desktop to sleep for 37.3% of the time. This is in addition to the existing savings, for total desktop sleep time of 72%. If we use the more aggressive default policy, where the desktop VM is migrated to the server unless there is UI activity, we find that LiteGreen puts desktop to sleep for 51.3% on time for a total



(a) Desktop sleep time (b) Desktop sleep time for user1 (c) Desktop sleep time for user2

Figure 7: Desktop sleep time from existing power management and LiteGreen's default and conservative policies

desktop sleep time of 86%.

The savings of the different approaches are also classified by day (8AM-8PM) and night (8PM-8AM) and also whether it was a weekday or a weekend. We note that substantial portion of LiteGreen desktop sleep time comes from weekdays, thereby highlighting the importance of exploting short idle intervals for energy savings.

8.2 Desktop Sleep Time for Selected Users

Based on the CPU utilization trace data, we found a user, say user1, who had bursts of significant activity separated by periods of no activity, likely because he/she manually switches off his/her machine when not in use. For this particular case, LiteGreen is unable to significantly improve on the energy savings of existing mechanisms (i.e., manual power management). This is reflected in the desktop sleep time for user1 in Figure 7b.

In contrast, for many of the users, say user2, the desktop exhibits low-levels of CPU activity with occasional spikes almost continuously, with only short gaps of inactivity. The few CPU utilization spikes can prevent Windows power management from putting the desktop to sleep, thereby wasting a lot of energy. However, LiteGreen is able to exploit this situation effectively, and puts the desktop to sleep for significantly longer time as shown in Figure 7c.

8.3 Server Resource Utilization during Consolidation

While the default policy provides higher savings than the conservative policy, it is clear that the default policy would stress the resources on the server more, due to hosting of more number of desktop VMs, than the conservative policy. We examine this issue next.

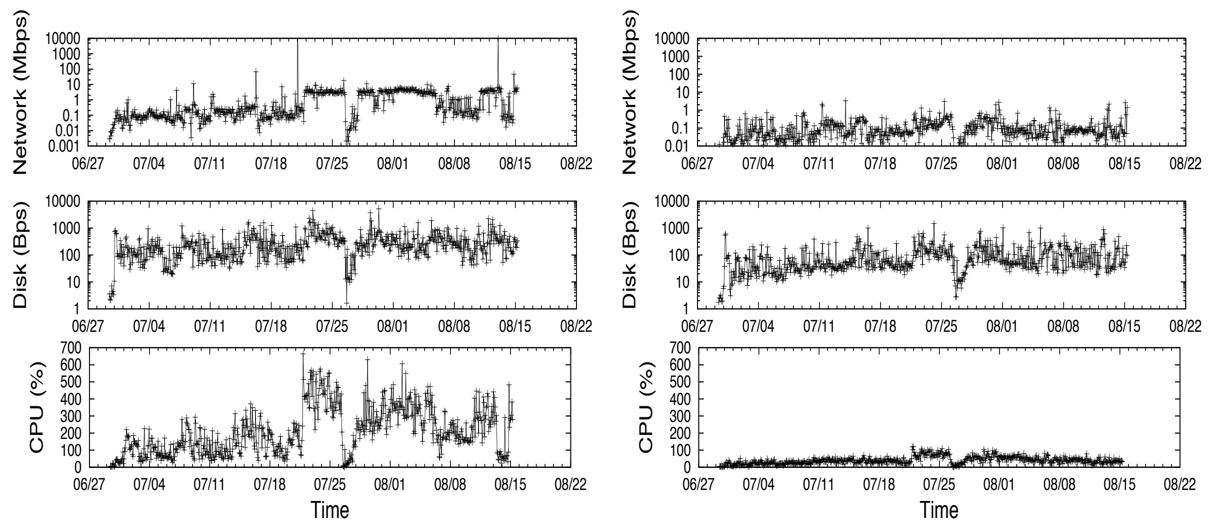
Figures 8a and 8b show the resource utilization due to idle desktop consolidation at the server for the default and conservative policies, respectively. The resources shown are CPU usage, bytes read/second from the disk, and network usage in Mbps.

First, consider CPU. Notice that the CPU usage at the server in the default policy spikes up to between $s_{pull} = 600$ an $s_{push} = 700$ but, as intended, never goes above s_{push}. In contrast, since the conservative policy pushes the VM back to the desktop as soon as $c_{push} = 20$ is exceeded, the CPU usage at the server hardly exceeds an utilization value of 100. Next consider disk reads. It varies between 10B-10KB/s for the default policy (average of 205 B/s) while it varies betwen 10B-1KB/s for the conservative policy (average of 41 B/s). While these numbers can be quite easily managed by the server, note that these are disk reads of idle, and not active, desktop VMs. Finally, let us consider network activity of the consolidated idle desktop VMs. For the default policy, the network traffic mostly varies between 0.01 to 10Mbps, but with occassional spikes all the way up to 10Gbps. In the case of conservative policy, the network traffic does not exceed 10Mbps and rarely goes above 1Mbps. While these network traffic numbers are manageable for a single server, these represent the workload of *idle* desktop machines. Scaling the server infrastructure to enable consolidation of *active* desktop VMs, as in the thin client model, will likely be expensive.

8.4 Energy Savings

We use calculations similar to the one performed in Section 7.3.3 for computing energy savings. Recall that power consumption of a desktop, without any energy savings mechanism, is simply (0.9*62.5+0.1*95) = 65.75 W per desktop.

Using existing energy saving mechanisms, where the desktop is put to sleep 35.2% of the time (Section 8.1), 0.352*(62.5-2.5) = 21.1 W per desktop or 32% of energy savings can be achieved. In the case of LiteGreen, our consolidation analysis (Section 8.3) suggests that one 8-core server is capable of hosting the idle desktops in the trace. Memory balooning results from Section 7.4 suggest that an idle VM could be packed in 384MB, implying that a 32GB server has enough memory capacity for up to to 96 idle VMs. Assuming some over-provisioning for capacity and redundancy, let us dedicate two servers



(a) Default policy

(b) Conservative policy

Figure 8: Resource utilization during idle desktop consolidation

of 250W each for the 120 desktops. The amortized cost of a server/desktop is then 500W/120 = 4.2W. Thus, power savings in LiteGreen (using the default policy with average desktop sleep time of 86%) is (0.86*(62.5 - 2.5) - 4.2) = 47.4 W per desktop or 72%, more than doubling the energy savings under existing mechanisms.

Finally, as in Section 7.3.3, if we were to include the amortized energy cost of centralized storage (2.1W/desktop), the energy savings in LiteGreen using the default policy is simply 47.4 - 2.1 = 45.3W per desktop or 69%.

9 Limitations and Future Work

We consider some limitations of LiteGreen, which also point to directions for future work.

9.1 Dependence on Shared Storage

Live migration assumes that the disk is shared between the source and destination machines, say in the form of network attached storage (NAS). This avoids the considerable cost of migrating disk content. However, this is a limitation of our current system since, in general, client machines would have a local disk, which applications (e.g., sharing of local files) need access to.

Recent work has demonstrated the migration of VMs with local virtual hard disks (VHDs) by using techniques such as pre-copying and mirroring of disk content [19] to keep the downtime to under 3 seconds in a LAN setting. Note that since the base OS image is likely to be already available at the destination node, the main cost is that of migrating the user data.

To quantify the costs involved in migrating the local disk, we performed detailed tracing of all file system operations on 3 actively used desktop machines using the `ProcessMonitor` tool [35]. Table 5 summarizes the

1-hour window (MB per hour)	4-hour window (MB per hour)
80-240	40-100

Table 5: Volume of dirty disk blocks

volume of dirty disk blocks that is generated, which represents the amount of disk state that would need to be migrated. We consider two cases: dirty blocks being pre-copied every hour versus every 4 hours. The latter provides a greater opportunity for temporal consolidation (i.e., merging of multiple writes to a block).

Migrating 100 MB of disk content over a GigE network would take 1.6 seconds, assuming an effective throughput of 500 Mbps. This means that over 2000 disk migrations can be supported per hour, which suggests that these migrations will not be the bottleneck. Further optimizations are possible, for instance, by transferring dirty data at a sub-block level and filtering out writes to scratch space.

Note that enterprise enviroments often employ network storage to hold persistent user data, since this enables the data to be backed up. In such a setting, the amount of data to be migrated would be further reduced to only temporary files generated by applications.

9.2 Heavyweightness

LiteGreen is a more heavyweight solution than the alternative proxy-based approach. To deploy LiteGreen, we would need to have desktop machines run a client hypervisor and also provision the necessary network bandwidth and server resources.

We believe that technology trends make it likely that the enterprise IT infrastructure would move in this direction. Virtualized desktops simplify management for the IT administrators. Also, the growth in thin clients would argue for server and network provisioning. Finally, the desire to support mobility and a "work from anywhere" capability would likely spur the development of a hybrid computing model wherein the desktop VM resides on the server when accessed from a thin client and migrates to the local machine at other times. Thus, we believe that that the LiteGreen approach fits in with these trends.

10 Conclusion

Recent work has recognized that desktop computers in enterprise environments consume a lot of energy in aggregate while still remaining idle much of the time. The question is how to save energy by letting these machines sleep while avoiding user disruption. LiteGreen uses virtualization to resolve this problem, by migrating idle desktops to a server where they can remain "always on" without incurring the energy cost of a desktop machine. The seamlessness offered by LiteGreen allows us to aggressively exploit short idle periods as well as long periods. Data-driven analysis of more than 65000 hours of desktop usage traces from 120 users as well as a small-scale deployment of LiteGreen on ten desktops, comprising 3200 user-hours over 28 days, shows that LiteGreen can help desktops sleep for 86-88% of the time. This translates to estimated desktop energy savings of 72-74% for LiteGreen as compared to 32% savings under existing power management mechanisms.

Acknowledgements

Rashmi KY helped with the desktop usage tracing effort at Microsoft Research India. Our shepherd, Katerina Argyraki, and the anonymous reviewers provided valuable feedback on the paper. We thank them all.

References

[1] AMD Vitualization Technology (AMD-V). `http://www.amd.com/us/products/technologies/virtualization/Pages/amd-v.aspx`.
[2] Citrix XenServer. `http://www.citrix.com/xenserver/`.
[3] Intel Vitualization Technology (VT-x). `http://www.intel.com/technology/itj/2006/v10i3/1-hardware/5-architecture.htm`.
[4] LiteGreen demo video. `http://research.microsoft.com/en-us/projects/litegreen/default.aspx`.
[5] Microsoft Hyper-V. `http://www.microsoft.com/windowsserver2008/en/us/hyperv-main.aspx`.
[6] QNAP SS-839 Pro Turbo Network Attached Storage. `http://www.qnap.com/pro_detail_hardware.asp?p_id=124`.
[7] Remote Desktop Protocol. `http://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx`.
[8] VMWare ESX Server. `http://www.vmware.com/products/esx/`.
[9] Wattsup Meter. `http://www.wattsupmeters.com`.
[10] Windows Management Instrumentation. `http://msdn.microsoft.com/en-us/library/aa394582(VS.85).aspx`.
[11] White Paper: Benefits and Savings of Using Thin Clients, 2X Software Ltd., 2005. `http://www.2x.com/whitepapers/WPthinclient.pdf`.
[12] White Paper: Wake on LAN Technology, June 2006. `http://www.liebsoft.com/pdfs/Wake_On_LAN.pdf`.
[13] Advanced Configuration and Power Interface (ACPI) Specification, June 2009. `http://www.acpi.info/DOWNLOADS/ACPIspec40.pdf`.
[14] Emerging Technology Analysis: Hosted Virtual Desktops, Gartner, Feb. 2009. `http://www.gartner.com/DisplayDocument?id=887912`.
[15] Worldwide PC 20092013 Forecast, IDC, Mar. 2009. `http://idc.com/getdoc.jsp?containerId=217360`.
[16] Agarwal, B., Akella, A., Anand, A., Balachandran, A., Chitnis, P., Muthukrishnan, C., Ramjee, R., and Varghese, G. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *USENIX NSDI* (Apr. 2010).
[17] Agarwal, Y., Hodges, S., Chandra, R., Scott, J., Bahl, P., and Gupta, R. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *NSDI* (Apr. 2009).
[18] Anand, A., Muthukrishnan, C., Akella, A., and Ramjee, R. Redundant in Network Traffic: Findings and Implications. In *ACM SIGMETRICS* (Seattle, WA, June 2009).
[19] Bradford, R., Kotsovinos, E., Feldmann, A., and Schioeberg, H. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. In *ACM VEE* (2007).
[20] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing energy and server resources in hosting centers. In *SOSP* (October 2001).
[21] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *NSDI* (May 2005).
[22] Das et al., T. LiteGreen: Saving Energy in Networked Desktops using Virtualization, Extended Version,. `http://research.microsoft.com/en-us/projects/litegreen/litegreen.pdf`.
[23] David, B. White Paper: Thin Client Benefits, Newburn Consulting, Mar. 2002. `http://www.thinclient.net/technology/Thin_Client_Benefits_Paper.pdf`.
[24] Dowty, M., and Sugerman, J. GPU Virtualization on VMware's Hosted I/O Architecture. In *USENIX WIOV* (2008).
[25] Garey, M. R., and Johnson, D. S. *Computers and intractability; a guide to the theory of NP-completeness*. W.H. Freeman, 1979.
[26] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI* (Dec. 2008).
[27] Kozuch, M., and Satyanarayanan, M. Internet Suspend/Resume. In *IEEE WMCSA* (June 2002).
[28] Madden, B. Understanding the role of client and host CPUs, GPUs, and custom chips in RemoteFX. `http://tinyurl.com/38wqson`.
[29] Moore, J., Chase, J., Ranganathan, P., and Sharma, R. Making Scheduling Cool: Temperature-aware Workload Placement in Data Centers. In *Usenix ATC* (June 2005).
[30] Nathuji, R., and Schwan, K. VirtualPower: Coordinated Power Management in Virtualized Enterprise Systems. In *SOSP* (Oct. 2007).
[31] Nedevschi, S., Chandrashekar, J., Liu, J., Nordman, B., Ratnasamy, S., and Taft, N. Skilled in the Art of Being Idle: Reducing Energy Waste in Networked Systems. In *NSDI* (Apr. 2009).
[32] Nordman, B. Networks, Energy, and Energy Efficiency. In *Cisco Green Research Symposium* (2008).
[33] Nordman, B., and Christensen, K. Greener PCs for the Enterprise. In *IEEE IT Professional* (2009), vol. 11, pp. 28–37.
[34] Reich, J., Kansal, A., Gorackzo, M., and Padhye, J. Sleepless in Seattle No Longer. In *USENIX ATC* (2010).
[35] Russinovich, M., and Cogswell, B. Process Monitor v2.8, 2009. `http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx`.
[36] Smith, J. M. A Survey of Process Migration Mechanisms. In *ACM SIGOPS Operating Systems Review* (July 1988), vol. 22, pp. 28–40.
[37] Srikantaiah, S., Kansal, A., and Zhao, F. Energy Aware Consolidation for Cloud Computing. In *HotPower* (Dec. 2008).
[38] Tolia, N., Wang, Z., Marwah, M., Bash, C., Ranganathan, P., and Zhu, X. Delivering Energy Proportionality with Non Energy Proportional Systems Optimizations at the Ensemble Layer. In *HotPower* (Dec. 2008).
[39] Waldspurger, C. Memory Resource Management in VMware ESX Server. In *OSDI* (Dec. 2002).
[40] Wood, T., Shenoy, P. J., Venkataramani, A., and Yousif, M. S. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI* (Apr. 2007).

Stout: An Adaptive Interface to Scalable Cloud Storage

John C. McCullough, John Dunagan[*], Alec Wolman[*], and Alex C. Snoeren

UC San Diego and [*]Microsoft Research, Redmond

ABSTRACT

Many of today's applications are delivered as scalable, multi-tier services deployed in large data centers. These services frequently leverage shared, scale-out, key-value storage layers that can deliver low latency under light workloads, but may exhibit significant queuing delay and even dropped requests under high load.

Stout is a system that helps these applications adapt to variation in storage-layer performance by treating scalable key-value storage as a shared resource requiring congestion control. Under light workloads, applications using Stout send requests to the store immediately, minimizing delay. Under heavy workloads, Stout automatically batches the application's requests together before sending them to the store, resulting in higher throughput and preventing queuing delay. We show experimentally that Stout's adaptation algorithm converges to an appropriate batch size for workloads that require the batch size to vary by over two orders of magnitude. Compared to a non-adaptive strategy optimized for throughput, Stout delivers over 34× lower latency under light workloads; compared to a non-adaptive strategy optimized for latency, Stout can scale to over 3× as many requests.

1. INTRODUCTION

Application developers are increasingly moving towards a software-as-a-service model, where applications are deployed in data centers and dynamically accessed by users through lightweight client interfaces, such as a Web browser. These "cloud-based" applications may run on hundreds or even thousands of servers to support hundreds of millions of users; the application servers in turn leverage high-performance scalable key-value storage systems, such as Google's BigTable [7] and Microsoft's Azure Storage [3], that allow them to gracefully handle variable client demand. Unfortunately, because these storage systems support many applications on a single shared infrastructure, they present application developers with a new source of variability: every application must now cope with a store that is being loaded by many applications' changing workloads.

Unlike variability in its own workload, which an application can easily monitor and often even predict, changes in the level of competition for shared storage resources are likely to be unexpected and outside the control of a particular application. Instead, each individual application must observe and react to changes in available storage-system throughput. Ideally, the collection of applications leveraging a particular scalable storage system would cooperate to achieve a mutually beneficial operating point that neither overloads the storage system nor starves any individual application.

Today, each application seeks to minimize its own perceived latency by sending each storage request immediately. Each storage request thus incurs overheads such as networking delay, protocol-processing, lock acquisitions, transaction log commits, and/or disk scheduling and seek time. However, when the store becomes heavily loaded, sending each request individually can lead to queuing at the store, and consequently high delay or even loss due to timeouts. In such heavily loaded situations, the throughput of the storage service can often be improved by batching multiple requests together, thereby reducing queuing delay and loss. Batching achieves this improvement by amortizing the previously mentioned overheads across larger requests, and prior work has documented that many stores provide higher throughput on larger requests [7, 16, 35].

Dynamically adjusting their degree of batching allows applications to achieve lower latency under light load and higher throughput under heavy load. Unfortunately, existing work applying control theory to computer systems offers no easily applicable solutions [18, 23]. For example, a common assumption in control theory is *modest actuation delay*: a reasonable and known fixed time between when an application changes its request rate and the store responds to this change. Scale-out key-value storage systems do not have such bounds, as an application can easily create a very deep pipeline of requests to the storage system. Other control theory techniques avoid this assumption, but bring other assumptions that are similarly unsatisfied by such storage systems. Instead, we observe that managing independent application demands in a scale-out key-value storage environment is quite similar to congestion control in a network: the challenge in both settings is determining an application's (sender's) "fair share." Moreover, the constraints of distributed congestion control—multiple, independent agents, unbounded actuation delay, and lack of a known bandwidth target—are quite similar to our own. Hence, we take inspiration from CTCP [37], a recently proposed delay-based TCP variant which updates send-rates based on deviation from the measured round-trip latency.

We propose an adaptive interface to cloud key-value storage layers, called Stout, that implements distributed congestion control for client requests. Stout works without any explicit information from the storage layer: its adaptation strategy is implemented solely at the application server (the storage client) and is based exclusively on the measured latency from unmodified scalable storage systems. This allows Stout to be more easily deployed, as individual cloud applications can adopt Stout without changing the shared storage infrastructure. Stout both adapts to sudden changes in application workload and converges to fairness among multiple, competing application servers employing Stout.

We show experimentally that Stout delivers good performance across a range of workloads requiring batching intervals to vary by over two orders of magnitude, and that Stout significantly outperforms any strategy using a fixed batching interval. Based on these results, Stout demonstrates that much of the benefit of adaptation can be had without needing to modify existing storage systems; to use a new store, Stout requires only internal re-calibration. By allowing cloud applications to sustain higher request rates under bursts, Stout can help reduce the expense of over-provisioning [8, 34]. Simultaneously, Stout provides good common case storage latency; this is critical to user-perceived latency because generating a user response often requires multiple interactions with the storage layer, thereby incurring this latency multiple times [11].

The primary novelty of Stout is its adaptive algorithm for dynamically adjusting the batching of storage requests. To better understand both the benefits and challenges in building an adaptive interface to shared cloud storage, we evaluate our adaptive control loop using a workload inspired by a real-world cloud service that is one component of Microsoft's Live Mesh cloud-based synchronization service [27]. In our performance evaluation, we demonstrate that: 1) Stout successfully adapts to a wide range of offered loads, providing under light workloads over 34× lower latency than a long fixed batching interval optimized for throughput, and under heavy workloads over 3× the throughput of a short fixed batching interval optimized for latency; 2) Stout provides

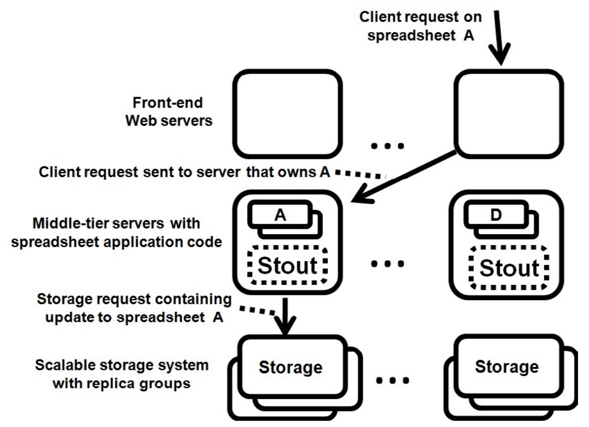

Client request on spreadsheet A
Front-end Web servers
Client request sent to server that owns A
Middle-tier servers with spreadsheet application code
A
D
Stout
Stout
Storage request containing update to spreadsheet A
Scalable storage system with replica groups
Storage
Storage

Figure 1: *Stout in a datacenter spreadsheet application.*

fairness without any explicit coordination across the different application servers utilizing a shared store; and 3) the same adaptation algorithm works well with three different cloud storage systems (a partitioned store that uses Microsoft SQL Server 2008; the PacificA research prototype [26]; and the SQL Data Services cloud store [30]).

2. BACKGROUND

Stout targets *interactive* cloud services. This class of services requires low end-user latency to a variety of data. Stout facilitates high-performance storage access for these services by controlling and adapting the way the services make use of back-end key-value storage systems to provide the best possible response time (i.e., minimize end-user latency). While we believe that Stout's general approach of using a control loop to manage the interactions with a persistent storage tier holds promise for many different kinds of cloud-based services, including those that process large data sets (e.g., services that use MapReduce [10] or Dryad [20]) the rest of this section elaborates on our current target class of interactive latency-sensitive cloud services.

Stout works with scalable services that are *partitioned*. A partitioned service is one that divides up a namespace across a pool of servers, and assigns "keys" within that namespace to only one server at a given point in time. To enable fast response times, the objects associated with the partition keys are stored in memory by these servers. Stout is responsible for handling all interactions with the back-end persistent storage tier. Figure 1 depicts a typical three-tier cloud service, and where Stout fits within that model. The first tier simply consists of front-end Web servers that route end-user requests to the appropriate middle-tier server; the middle tier contains the application logic glued together with Stout, and the back-end tier is a persistent storage system.

As a concrete example, consider an online spreadsheet application, such as that provided by Google Docs [15]. The user-interface component of the spreadsheet application runs inside the client Web browser. As users perform actions within the spreadsheet, requests are submitted to the cloud infrastructure that hosts the spreadsheet service. User requests arrive at front-end Web servers after traversing a network load balancer, and the front-end server routes the user request to the appropriate middle-tier server which holds a copy of the spreadsheet in memory. Each server in the middle tier holds a large number of spreadsheets, and no spreadsheet is split across servers. Whenever the processing of a user request results in a modification to the spreadsheet, the changes are persisted to a scalable back-end storage system before the response is sent back to the user.

Many of today's Web services are built using the same paradigm as the spreadsheet application above. For example, a service for tracking Web advertising impressions can store many "ad counters" at each middle-tier server. Email, calendar, and other online office applications can also use this partitioning paradigm [15, 19, 29].

Forcing writes to stable storage before responding to the user ensures strong consistency across failures in the middle tier; that is, once the user has received a response to her request to commit changes, she can rest assured they will always be reflected by subsequent reads. So long as a middle-tier server maintains these semantics, it is free to optimize the interactions with the storage layer. Thus, when a middle-tier is handling multiple changes, it can batch them together for the storage layer.

3. ADAPTIVE BATCHING

Batching storage requests together before sending them to the store leads to several optimization opportunities (Section 3.1). However, delaying requests to send in a batch is only needed when the store would otherwise be overloaded; if the store is lightly loaded, delaying requests yields a net penalty to client-visible latency. This motivates Stout's adaptation algorithm, which measures current store performance to determine the correct amount of batching as workloads change (Section 3.2).

3.1 Overlapped Request Processing

Having multiple storage requests to send in a batch requires the application to overlap its own processing of incoming client requests. Figure 2 illustrates overlapped request processing for both reads and writes. Note that only reads that miss the middle-tier's cache require a request to the store; cache hits are serviced directly at the middle-tier. Initially, the application receives two client requests, "Change 1 on A" and "Change 2 on B". Both of the client requests are processed up to the point that they generate requests for the store. These are then sent in a single batch to the store. After the store acknowledgment arrives, replies are sent to both of the client requests. While waiting for the store acknowledgment, client re-

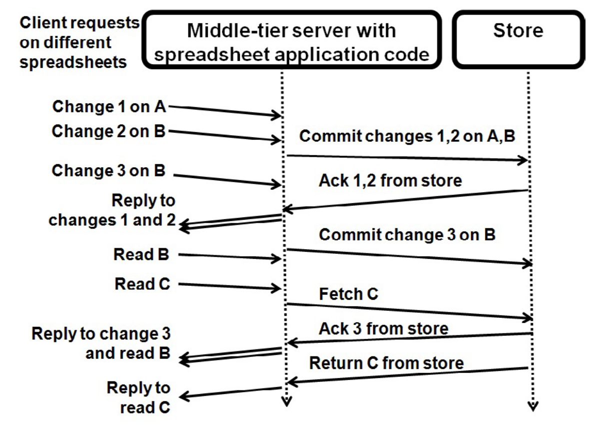


Figure 2: *An example of overlapped request processing.*

quest "Change 3 on B" arrives and is processed up to the point of generating a request to the store. Later, client request "Read B" arrives and hits the middle-tier cache, while "Read C" arrives and requires fetching C from the store. We describe in Section 4 how the Stout implementation handles the multiplexing of these storage requests into batches and the corresponding de-multiplexing of store responses. Grouping storage requests together enables two well-known optimizations:

- **Batching:** Many stores perform better when a set of operations is performed as a group, and many systems incorporate a group-commit optimization [6, 16, 17]. The performance improvements arise from a number of factors, such as reducing the number of commit operations performed on the transaction log, or reducing disk seek time by scheduling disk operations over a larger set. Storage system performance further improves by initiating batching from the middle-tier for reasons that include reduced network and protocol processing overheads.

- **Write collapsing:** When multiple writes quickly occur on the same object, it can be significantly more efficient for the middle-tier server to send only the final object state. An example where write collapsing may arise in cloud services is tracking advertising impressions, where many clients may increment a single counter in quick succession and the number of writes can be safely reduced by writing only the final counter value to the store. Many workloads possess opportunities for write collapsing, and many prior systems are designed to exploit these opportunities [36, 40].

Stout's novelty is managing how these optimizations are exploited for a shared remote store based on a multiplicative-increase multiplicative-decrease (MIMD) control loop. It does this by varying a single parameter, the batching interval. At the end of each interval, Stout sends all writes and cache-miss reads to the store. In this way, the batch size is simply all such reads and writes

		Batching Interval	
	No batching	10ms	20ms
Requests/second	11k	13k	17k
Throughput Gain	-	18%	55%

Table 1: *How a service's maximum throughput can increase by exploiting batching.*

EWMA factor	1/16
$thresh$	0.85
$MinRequests$	10
$MinLatencyFrac$	1/2

Table 2: *Parameters to make measurements and comparisons robust to jitter.*

generated in the previous interval, and write collapsing is obtained to the extent that multiple updates to the same key happened during this interval. Pipelining occurs if this batch is sent to the store while an earlier batch is still outstanding (i.e., when the batching interval is less than the store latency).

For a given workload, a longer batching interval will allow more requests to accumulate, leading to a larger batch size and potentially greater throughput. Table 1 quantifies the improvement in maximum throughput for one of the experimental configurations that we use to evaluate Stout. This configuration is described in detail in Section 5.2. Our goal here is simply to convey the magnitude of potential throughput gain (over 50%) from even slightly lengthening the batching interval. This throughput gain translates into a much larger set of workloads that can be satisfied without queues building up at the store and requests eventually being dropped.

However, the improved throughput of a longer batching interval is not always needed; if the workload is sufficiently light, client latency is minimized by sending every request to the store immediately. For example, the batching intervals that lead to the higher throughput shown in Table 1 also add tens of milliseconds to latency. To determine the right batching interval at any given point in time, Stout measures the current performance of the store. Stout uses these measurements to set its batching interval to be shorter if the store is lightly loaded, and longer if the store is heavily loaded.

3.2 Updating the Batching Interval

The problem of updating the batching interval is a classic congestion control problem: competing requests originate independently from a number of senders (i.e., middle-tiers); these requests have to share a limited resource—the store—and there is some delay before resource oversubscription is noticed by the sender (in this case, the time until the store completes the request). Like TCP, Stout does not require explicit feedback about the degree of store utilization. This allows Stout to be easily deployed with a wide range of existing storage systems. Unlike TCP, Stout must react primarily to delay rather than loss, as stores typically queue extensively before dropping requests. Thus, our design for Stout's control loop borrows from a recent delay-based TCP, Compound TCP (CTCP [37]). In general, delay-based TCP variants react when the current latency deviates from a baseline, falling back to traditional TCP behavior in the event of packet loss. Compared to TCP Vegas [5] (another delay-based TCP), CTCP more rapidly adjusts its congestion window so that it can better exploit high bandwidth-delay product links. For Stout, rapid adjustment means faster convergence to a good batching interval.

However, one aspect of our problem differs from that addressed by congestion control protocols. Delay-based TCP assumes that increasing delay reflects congestion and will consequently reduce the sending rate to alleviate that congestion. Stout acts to reduce congestion by improving per-request performance rather than reducing send rates. Increasing the batch size means that the next request will take longer to process even in the absence of congestion. Furthermore, Stout must distinguish this increased delay due to an increased batch size from increased delay due to congestion. For this reason, Stout has to depart from CTCP by incorporating throughput, not just delay, into measuring current store performance and assessing whether the store is congested.

The remainder of this section describes Stout's approach to updating the batching interval, which we denote by $intrvl$, the time in milliseconds between sending batches of requests to the store. In Section 3.2.1, we describe how Stout decides when it is time to update the batching interval. In Section 3.2.2, we describe how Stout decides whether to increase or decrease the batching interval. Increasing the batching interval corresponds to backing off—going slower because of the threat of congestion—while decreasing the batching interval corresponds to accelerating. Then in Section 3.2.3, we describe how Stout decides how much to increase or decrease the batching interval.

3.2.1 When to Back-off or Accelerate

Like TCP and its many variants, Stout is self clocking: it decides whether or not to back-off more frequently when the store is fast, and less frequently when the store is slow. To this end, Stout tracks the latency between when it sends a request to the store and when it receives a response. Stout computes the mean of these latencies over every request that completes since the last decision to adjust $intrvl$; we abbreviate the mean latency as lat.

Stout decides to either back-off or accelerate as soon as both $MinRequests$ requests have completed and

$(MinLatencyFrac \times lat)$ time has elapsed; the former term is dominant when there is little pipelining, and the latter term is dominant when there is significant request pipelining. We find that this waiting policy mitigates much of the jitter in latency measurements across individual store operations. Table 2 shows the settings for these parameters that we used in our experiments, as well as the other parameters (introduced later in this section) that play a role in making Stout robust to jitter.

3.2.2 Whether to Back-off or Accelerate

Stout makes its decision on whether to back-off or accelerate by comparing the current performance of the store to the performance of the store in the recent past. We denote the store's current performance by $perf$, its recent performance by $perf^*$, and we explain how both are calculated over the next several paragraphs. As mentioned in the Introduction, Stout restricts its measurements to response times so that it can be re-used on different stores, as this measurement requires no store-specific support. The performance comparison is done with some slack (denoted as $thresh$), so as to avoid sensitivity to small amounts of jitter in the measurements:

```
if ($perf < (thresh \times perf^*)$)
    BACK-OFF
else
    ACCELERATE
```

We calculate $perf$ using the number of bytes sent to and received from the store during the most recent self-clocking window (denoted by $bytes$), the mean latency of operations that completed during this same period of time, and the length of the current batching interval. (Note that higher $perf$ is better.)

$$perf = \frac{bytes}{lat + intrvl}$$

Our $perf$ definition is a simple combination of latency and throughput: Stout's latency is $intrvl + lat$, the time until Stout initiates a batch plus the time until the store responds; Stout's throughput is $bytes/intrvl$, the amount sent divided by how often it is sent.

Incorporating throughput appropriately rewards backing-off when it causes throughput to increase and the throughput improvement outweighs the larger store latency (lat) from processing a larger batch. By contrast, just measuring latency could lead to an undesirable feedback loop: Stout could back-off (taking more time between batches), each batch could send more work and hence take longer, the store would appear to be performing worse, and Stout could back-off again.

Stout must compute recent performance ($perf^*$) in a manner that is robust to background noise, is sensitive to the effects of Stout's own decisions, and that copes with delay between its changes and the measurement of those changes. To this end, Stout computes $perf^*$ over different sets of recent measurements depending on its own recent actions (e.g., backing off or accelerating). To explain the $perf^*$ computation, we first present the algorithm and then provide its justification.

```
if (last decision was ACCELERATE)
    $perf^* = \text{MAX}_i(\frac{bytes_i}{lat_i + intrvl_i})$        (1)
else // last decision was BACK-OFF
    if ($intrvl < \text{EWMA}(intrvl_i)$)
        $perf^* = \frac{\text{EWMA}(bytes_i)}{\text{EWMA}(lat_i) + \text{EWMA}(intrvl_i)}$        (2)
    else // ($intrvl \geq \text{EWMA}(intrvl_i)$)
        $perf^* = \frac{\text{EWMA}(bytes_i)}{\text{EWMA}(lat_i) + intrvl}$        (3)
```

Equation (2) for computing $perf^*$ is the most straightforward: it is an exponentially weighted moving average (EWMA) over all intervals i since the last acceleration. However, Stout cannot always wait for latency changes to be reflected in this EWMA because of the risk of overshooting—not reacting quickly enough to latency changes that Stout itself is causing. This risk motivates Equations (1) and (3), which we now discuss.

Equation (1) prevents overshoot while accelerating. When Stout is accelerating, it runs a risk of causing the store to start queuing. To prevent this, Stout heightens its sensitivity to the onset of queuing by computing recent performance ($perf^*$) as the best performance since the last time Stout backed-off. Stout stops accelerating as soon as current performance drops behind this best performance. By contrast, calculating recent performance using an EWMA would mask any latency increase due to queuing until it had been incorporated into the EWMA multiple times.

Equation (3) prevents overshoot while backing-off: when Stout backs off, the increase in $intrvl$ can penalize current $perf$, potentially causing Stout to back-off yet again, even if throughput (the $bytes/lat$ portion of $perf$) has improved. To address this, when the current $intrvl$ is larger than its recent history, we use it in calculating both $perf$ and $perf^*$.

3.2.3 How Much to Back-off or Accelerate

Stout reuses the MIMD-variant from CTCP [37]: MIMD allows ramping up and down quickly, and as in CTCP, incorporating $\sqrt{intrvl}$ into the update rule provides fairness between competing clients. A minor difference between CTCP and Stout is that CTCP modifies the TCP window, and backing-off corresponds to decreasing this window; Stout modifies its batch interval, and backing-off corresponds to increasing this interval.

Stout backs off using a simple multiplicative back-off step, and it accelerates using a multiplicative factor that decreases as $intrvl$ approaches its lower limit (1 ms in

α'	1/400
α_{max}	1/2
β	1/10
$intrvl_{initial}$	80 ms
$intrvl_{max}$	400 ms

Table 3: *Parameters for gain and boundary conditions. These parameters are analogous to those in CTCP, e.g., $intrvl_{max}$ corresponds to RTO_{max}.*

this case):

BACK-OFF:

$$intrvl_{i+1} = (1 + \alpha) * intrvl_i$$

ACCELERATE:

$$intrvl_{i+1} = (1 - \beta) * intrvl_i + \beta * \sqrt{intrvl_i}$$

Competing clients converge to fairness because slow clients accelerate more than fast clients when the store is free, and all clients back-off by an equal factor when the store is busy. The CTCP paper formally analyzes this convergence behavior [37].

The incremental benefit of additional batching decreases as the batch size grows. Because of this, Stout must react more dramatically if the store is already processing large batches and then starts to queue. To accomplish this, we make α (the back-off factor) proportional to an EWMA of latency, with an upper bound:

$$\alpha = \text{MAX}(\text{EWMA}(lat_i) * \alpha', \alpha_{max})$$

Finally, stores occasionally exhibit brief pauses in processing, leading to short-lived latency spikes (this behavior is described in greater detail in Section 5.7). This behavior could cause Stout to back-off dramatically, and then take a long time recovering. To address this, we introduce an $intrvl_{max}$ parameter; just as TCP will never assume that the network has gotten so slow that retransmissions should wait longer than RTO_{max}, Stout will never assume that store performance has degraded to the point that batches should wait longer than $intrvl_{max}$. This bounds Stout's operating range, but allows it to recover much more quickly from brief store pauses.

Table 3 shows the gain and boundary condition parameter settings. As in CTCP's parameter settings, the initial batching interval is conservative, and the gain parameters lead to bigger back-offs than accelerations, similar to how TCP backs off faster than it accelerates. Our experiments in Section 5 show that Stout works well with these choices, and that it effectively converges to batching intervals spanning over two orders of magnitude.

4. IMPLEMENTATION

Stout's primary novelty is its algorithm for dynamically adjusting the batching of storage requests. We implement the Stout prototype to evaluate this algorithm

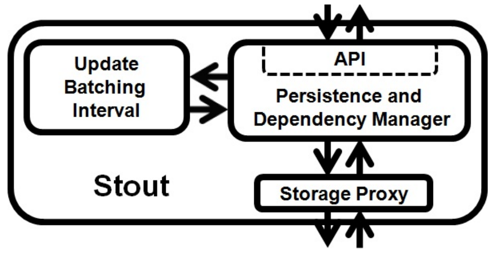
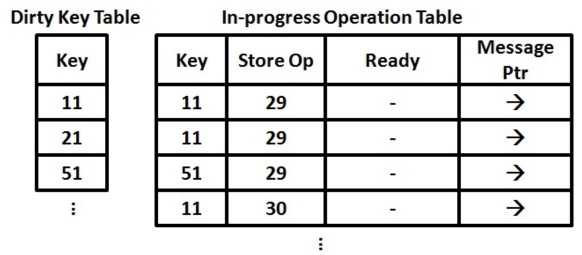


Figure 3: *The internal architecture of Stout.*



Dirty Key Table

Key
11
21
51
⋮

In-progress Operation Table

Key	Store Op	Ready	Message Ptr
11	29	-	→
11	29	-	→
51	29	-	→
11	30	-	→
⋮			

Figure 4: *Data structures for Stout's dependency map.*

with a real-world cloud service (a component of Microsoft's Live Mesh service [27]). We first describe how the application ensures that each key is owned by a single middle-tier (Section 4.1). We then describe the Stout internal architecture (Section 4.2), followed by how Stout multiplexes storage requests into batches and the corresponding de-multiplexing of store responses (Section 4.3). Finally, we describe the Stout API by walking through an example of its use (Section 4.4).

4.1 Key Ownership

As discussed previously, applications that use Stout must ensure that all requests on a given partition key are handled by only one middle-tier server at any given point in time. In particular, the write collapsing optimization requires that all updates to a given partition key are being sent to the same server. This requirement could be met using a variety of techniques; the applications we evaluate rely on Centrifuge [2].

Centrifuge is a system that combines lease-management with partitioning. Centrifuge uses a logically centralized manager to divide up a flat namespace of keys across the middle-tier servers. Centrifuge grants leases to the middle-tiers to ensure that responsibility for individual objects within the namespace are assigned to only one server at any given point in time. Front-end Web servers route requests to middle-tiers via Centrifuge's lookup mechanism.

4.2 Stout Internal Architecture

Stout's internal architecture divides the problem of managing interaction with the store into three parts, as depicted in Figure 3. The "Persistence and Dependency

Stout.Fill(key)	Ask Stout to fetch objects associated with partition key from store.
Stout.MarkDirty(key)	Mark objects associated with partition key as modified, so that Stout knows to persist them.
Stout.MarkDeleted(key)	Mark objects associated with partition key as deleted, so that Stout knows to delete them from store.
Stout.SendMessageWhenSafe(key, sendMsgCallback)	Sends a reply message after Stout's internal dependency map indicates it is safe to send response.
Stout.SerializeDone(key[], byte[][])	App indicates completion of Stout's request to serialize objects.
App.Serialize(key[])	Callback invoked by Stout for objects that have been marked dirty. Requests App to convert objects into byte arrays to send to the store and respond with *SerializeDone()*.
App.Deserialize(key[], byte[][])	Callback invoked by Stout when *Fill()* responses arrive from store. Converts each byte[] into object.

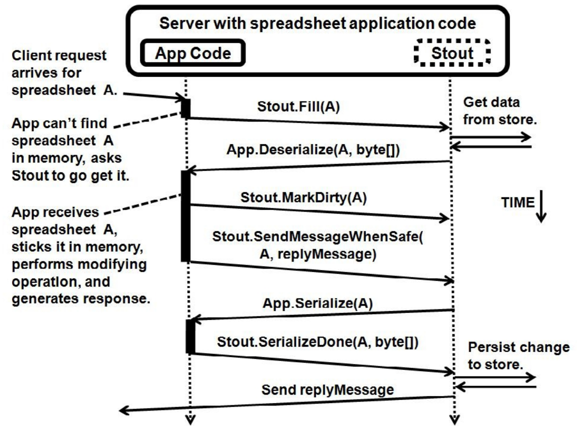
Table 4: *Client API. All calls are asynchronous.*

```
// We have received a message containing ``update''
// for the spreadsheet named by ``key''.
ProcessRequest(update, key) {
  // If we don't have the state for this key,
  // we ask Stout to get it from the back-end store.
  if (table[key] == null)
    Stout.Fill(key);
  ... // Block until Stout has filled in table[key].

  // Spreadsheet-specific logic is in UpdateSheet().
  replyMessage = UpdateSheet(table[key], update);
  Stout.MarkDirty(key); // Tell Stout about update.

  // Ask Stout to send reply when update is persisted.
  Stout.SendMessageWhenSafe(key, replyMessage);
}
```

(a) *Placement of API calls in sample application code. Stout and the application communicate via message passing, so the application does not need to coordinate its locking with Stout.*


Server with spreadsheet application code
App Code
Stout
Client request arrives for spreadsheet A.
App can't find spreadsheet A in memory, asks Stout to go get it.
App receives spreadsheet A, sticks it in memory, performs modifying operation, and generates response.
Stout.Fill(A)
Get data from store.
App.Deserialize(A, byte[])
Stout.MarkDirty(A)
TIME
Stout.SendMessageWhenSafe(A, replyMessage)
App.Serialize(A)
Stout.SerializeDone(A, byte[])
Persist change to store.
Send replyMessage

(b) *Flow of calls between spreadsheet application, Stout and store. The portion of time when the spreadsheet application is active is denoted by the thick black line.*

Figure 5: *An example use case of a spreadsheet application interacting with Stout.*

Manager" component handles correctness and ordering constraints (e.g., ensuring that requests are committed to the store before replies are sent), as described in Section 4.3. Applications interact with this component through the API described in Section 4.4. The "Update Batching Interval" component implements the adaptive batching algorithm from Section 3.2. The "Storage Proxy" component is a thin layer that connects Stout to a specific scalable storage system. We have implemented three proxies to interface Stout with different cloud storage systems, and all three use TCP as a transport layer.

4.3 Persistence and Dependencies

Each middle-tier uses Stout to manage its in-memory data as a coherent cache of the store. Stout is responsible for communicating with the store and ensuring proper message ordering. The application is then responsible for calling Stout when it: (1) needs to fetch data from the store, (2) modifies data associated with a partition key, or (3) wants to send a reply to a client.

Stout ensures proper message ordering by maintaining a dependency map that consists of two tables, as depicted in Figure 4. Keys are added to the table of dirty keys whenever the application notifies Stout that a key has been modified. Messages provided by the application are added to the table of in-progress operations if the key is dirty or there are any outstanding operations to the store on this key; otherwise, the messages are sent out immediately. When Stout sends a batch of writes to the store to commit the new values of some keys, those same keys are removed from the table of dirty keys, and Stout fills in the "Store Op" for the appropriate rows in the table of in-progress operations. When a store operation returns, Stout sends out messages in the order they were received from the application.

Figure 4 depicts both batching (keys 11 and 51 were both sent in storage operation 29) and write collapsing (two update operations for key 11 were both conveyed in operation 29). Stout requires the store to commit operations in order, but the store may still return acknowledgments out of order. In our example, if the acknowledgment of 30 arrives before the acknowledgment of 29, Stout would mark the fourth row of the table "Ready" and send the message once all earlier store operations on key 11 are ready and their messages sent.

4.4 Stout API

Table 4 describes each of the API calls and the callbacks that applications must provide for Stout. Figure 5a shows how a datacenter spreadsheet application places the API calls in its code. Before the application's *ProcessRequest()* function is called, the application has already received the request, done any necessary authentication, and checked that it holds the lease for the given partition key. *ProcessRequest()* handles both modifying spreadsheet objects (done in *UpdateSheet()*) and interacting with Stout: using Stout to fetch state from the store, letting Stout know that the state has been updated, and telling Stout about a reply that should be sent once the update has been persisted to the store. We do not show the code to send the reply, but note that before the application sends the reply message to the client, it must check that the lease for the partition key has been continuously held for the duration of the operation.

Figure 5b illustrates the ordering of calls between the application and Stout, and between Stout and the store. When an application or service first receives a request on a given partition key, it fetches the state associated with that partition key using the *Stout.Fill()* call. When the state arrives, Stout calls *App.Deserialize()* to create in-memory versions of fetched objects, which can then easily be operated on by the application logic.

To support coherence, Stout needs to know when operations modify internal service state, so that these updates can be saved to the store. Since Stout has no *a priori* knowledge of the application internals, Stout requires the service developer to call *Stout.MarkDirty()* in any service methods that modify objects associated with a partition key. At some point after a key has been marked as dirty, the Stout persistence manager will call *App.Serialize()* on a set of dirty keys. By delaying calls to *App.Serialize()*, Stout allows modifications to the same object to overwrite each other in-memory, thus capturing write collapsing. The application then responds by calling *Stout.SerializeDone()* with the corresponding byte arrays to be sent to the store.

When a Stout-enabled service would like to send a response to a user's request, it must use *Stout.SendMessageWhenSafe()* to provide the outgoing message callback to Stout. Stout will then take responsibility for determining when it is safe to send the outgoing message, based on its knowledge of the current interactions with the persistent store related to the partition key for that request. For example, if the message is dependent on state which has not yet been committed to the persistent store, Stout cannot release the message until it receives a store acknowledgment that the commit was successful.

For certain services, the state associated with a partition key may be large enough that one does not want to serialize the entire object every time it is modified, especially if the size of the modifications is small compared to the size of the entire state. To handle this case, the API supports an additional parameter, a sub-key. Stout keeps track of the set of dirty sub-keys associated with each partition key, and asks the application for only the byte arrays corresponding to these sub-keys. Finally, Stout also enables deletion from the persistent store using the *Stout.MarkDeleted()* call, which similarly takes both partition keys and sub-keys. Stout tracks these requested deletes, and then includes them in the next batch sent to the store, along with any read and write operations.

5. EVALUATION

We now demonstrate the benefits of Stout's adaptation strategy. In Section 5.1, we describe the setup for our experiments. In Section 5.2, we evaluate the potential benefits of batching and write collapsing in the absence of adaptation. In Sections 5.3-5.6, we evaluate Stout's adaptation strategy and show that it outperforms fixed strategies with both constant and changing workloads, that multiple instances of Stout dynamically converge to fairly sharing a common store, and that Stout's adaptation algorithm works across three different cloud storage systems. Finally, in Section 5.7, we examine the behavior of our store, and we show that Stout is robust to brief "hiccups" where the store stops processing requests.

5.1 Experimental Setup

We first describe the application that we ported to use Stout and this application's workload, and we then characterize the system configuration for our experiments.

5.1.1 Application and Workload

The application we run on our middle-tier servers is a "sectioned document" service. This service is currently in production use, and additional details can be found in the Centrifuge paper [2]. This service allows documents to contain independent sections that can be named, queried, added, and removed. The unmodified service is approximately 7k commented lines of C# code, and we ported this service to use the Stout API changing approximately 300 lines of code. Stout itself consists of 4k commented lines of code and the storage proxies are each approximately 600 commented lines of code.

In production, this service is deployed on multiple large pools of machines. One pool is used exclusively to store device presence: a small amount of addressing information, such as IP address, and an indication whether the device is online. Although we were unable to obtain a trace from production, we used known characteristics of the production system to guide the design of a synthetic client workload for our evaluation: varying request rates on a large number of small documents, 2k documents per middle-tier, each consisting of a single 256-byte section. At saturation, our store is limited by the total number of operations rather than the total number of bytes being stored under this workload, a common situation [7, 31].

In this synthetic workload, we designed the read/write mixture to best evaluate Stout's ability to adapt under workload variation. We avoid making the workload dominated by reads, because this would have primarily loaded the middle-tiers, and Stout's goal is to appropriately adapt when the store is highly loaded. We also avoided a pure-write workload because this would not capture how reads that hit the middle-tier cache are delayed if they touch documents that have been updated but where the update has not yet been committed to the store. This led us to choose a balanced request mixture of 50% reads and 50% writes.

In the commercial cloud service that motivates our workload, all data fits in RAM—Stout is using the store for persistence, not capacity. Because of this, read latencies are uniformly lower than write latencies (e.g., Figure 9 in Section 5.3). In the Stout consistency model, write latencies impact the user experience because responses are only sent after persisting state changes (e.g., after saving a spreadsheet update). Because writes form the half of the workload that poses the greater risk of poor responsiveness, the rest of the evaluation reports only write latencies unless otherwise noted.

5.1.2 System Configuration

Our testbed consists of 50 machines with dual-socket quad-core Intel Xeon 5420 CPUs clocked at 2.5 GHz, with 16 GB of RAM and 2×1 TB SATA 7200 rpm drives. We chose the ratio of front-ends to middle-tiers to storage nodes such that the overall system throughput was maximized subject to the constraint that the storage system was the bottleneck. This led to dividing the 50 machines into 1 experiment controller, 1 Centrifuge lease manager, 12 front-ends that also generate the synthetic client workload, 32 middle-tiers using the Stout library, and 4 systems running the persistent storage system. The choice of 32 middle-tiers means there are 64k total documents in the system. Unless noted otherwise, latency is measured from the front-ends (denoted FE latency in the figures)—this represents the part of end-to-end client latency due to the datacenter application.

Most of our experiments run Microsoft SQL Server 2008 Express on each of the four storage servers to implement persistent storage. We configure the storage servers to use a dedicated disk for SQL logging, and we followed the SQL documentation to ensure persistence under power loss, including disabling write-caching on our SATA drives [12]. The Stout storage proxy consists of a simple client library that performs hash-based partitioning of the database namespace. For a small number of experiments, we used two additional stores: the PacificA storage system [26] which uses log-based storage and replication, and the commercially available SQL Data Services (SDS) cloud-based storage system [30].

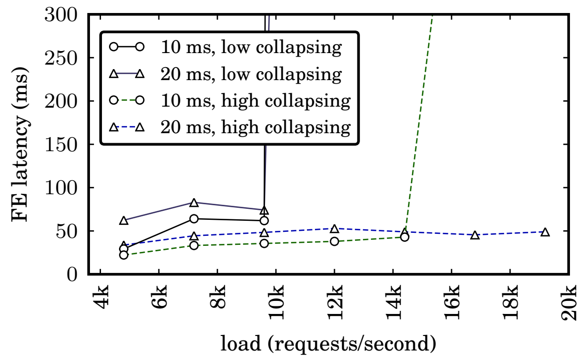


Figure 6: *Two fixed batching intervals (10 ms, 20 ms) on a workload with low write collapsing (10k documents) or high write collapsing (100 documents).*

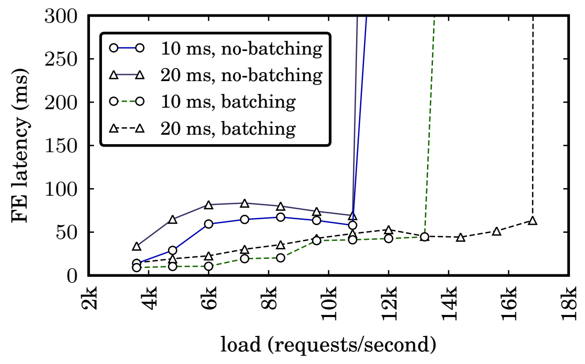


Figure 7: *Two fixed batching intervals (10 ms, 20 ms) on identical workloads with and without batching.*

Under our workload, these stores occasionally exhibit brief hiccups where they pause in processing; we describe this in more detail in Section 5.7. Unless noted otherwise, we report data from runs without hiccups.

5.2 Batching and Write Collapsing

We perform two experiments to evaluate the potential performance improvements that are enabled by the batching and write collapsing optimizations. For both experiments, we use two different fixed batching intervals—10 and 20 ms—to isolate the benefits of batching and write collapsing from adaptation.

Figure 6 shows the performance benefits of write collapsing. For this experiment, requests are delayed for the duration of the batching interval, but they are not actually sent in a batch; at the end of each batching interval, all the accumulated requests are sent individually to the store. Because of this, the entire observed performance difference is due to write collapsing. The low collapsing workload consists of 10k documents spread across the 32 middle-tiers, while high collapsing consists of only 100 documents, significantly increasing the probability that there are multiple updates to the same document within the batching interval. The graph shows that, as expected, write collapsing reduces latency and improves

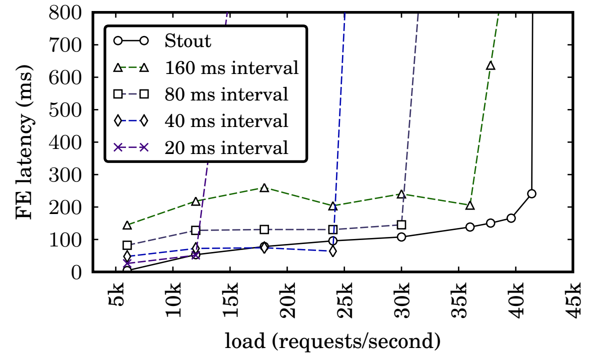


Figure 8: *Mean response latency for writes: Stout versus fixed batching intervals over a wide variety of loads.*

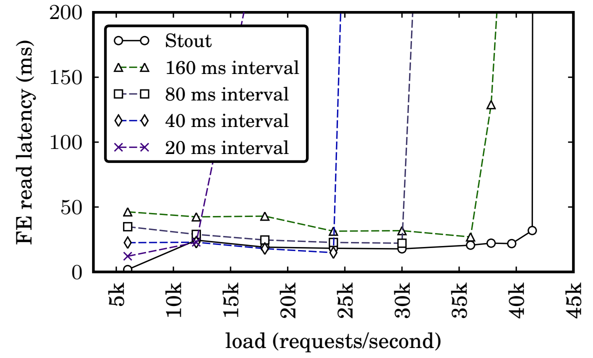


Figure 9: *Mean response latency for reads: Stout versus fixed batching intervals over a wide variety of loads. Note that the y-axis is 4× smaller than in Figure 8.*

the capacity of the system. For the low collapsing case, we see that the 10- and 20-ms batching intervals can satisfy between 4k requests/second and 10k requests/second with better client perceived latency for a 10-ms batching interval. However, at 12k requests/second the storage system is overloaded, resulting in a large queuing delay represented by an almost vertical line. In contrast, for the high collapsing workload a 10-ms batching interval can sustain nearly 15k requests/second because the actual number of writes sent to the store is reduced. For the 20-ms batching interval, the number of writes is reduced enough to shift the bottleneck from the store to the middle-tier and provide up to 80k requests/second.

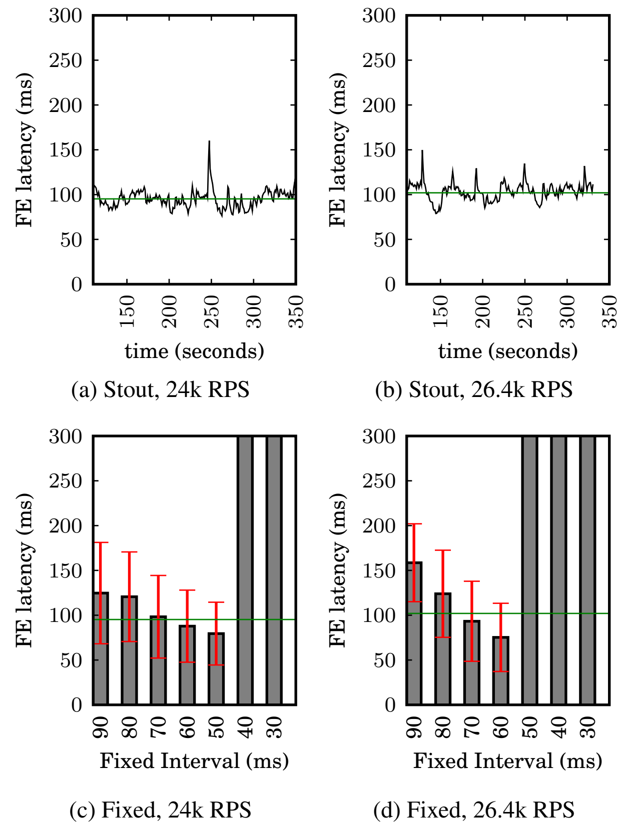
Figure 7 shows the performance benefits of batching. The no-batching experiments reflect disabling batching using the same methodology as in the write collapsing experiment: requests are delayed but then sent individually. We see that the throughput benefits of batching are noticeable at 10 ms, and they increase as the batching interval gets longer, which in turn causes the batch size to get larger. At a 20-ms batching interval, batching allows the system to handle an additional 6k requests per second. The amount of write collapsing for each fixed batching interval in this experiment is constant (and small). We separately observed that PacificA also delivers performance benefits from batching (this is detailed in Section 5.6, where we evaluate Stout on both PacificA and SDS). As mentioned in the Introduction, the reason for batching's benefits depend on the individual store being used; for our partitioned store built on SQL, we separately determined that a significant portion of the benefit comes from submitting many updates as part of a single transaction.

5.3 Adaptive vs. Fixed Batching

In this section, we demonstrate that Stout is effective across a wide operating range of offered loads, and investigate the overhead imposed by Stout's adaptation over the *best* fixed batching interval at a given load.

Figures 8 and 9 compare Stout to fixed batching intervals that vary from 20 ms up to 160 ms, for offered loads that range from 5k requests/second all the way up to 41k requests/second, which is very near the maximum load that our storage system can support. These figures are generated from the same experiments: Figure 8 shows the mean response latency for write operations whereas Figure 9 shows the latency for reads – all reads are cache hits in this workload, but the latency numbers do include delay from reading an updated document where the update has not yet been committed to the store. In both graphs, we see that Stout provides a wider operating range than any of the fixed batching intervals, and it provides response latencies that are either similar to or better than the fixed batching intervals. Looking at the two extremes of latency and throughput in Figure 8, Stout's 4.2 ms latency at 6k requests/second is over 34× smaller than the 144 ms latency incurred by the longest fixed batching interval in this experiment (160 ms), while Stout's 41k requests/second maximum is over 3× larger than the 12k requests/second maximum for the shortest fixed batching interval in this experiment (20 ms).

To understand the overhead of Stout's adaptation, we compare Stout to different fixed batching intervals at fine granularity under two fixed workloads. In Figures 10 (a) and (b), the time series show Stout's latency to be relatively steady, and for this reason we focus on the mean latency throughout this section. Figure 10 (c) compares Stout's mean to fixed intervals with an offered load of 24k requests/second. The best fixed interval is at 50 ms, and here we observe that Stout's adaptation adds just under 15 ms to the response latency (from 80 to 94 ms) and is within the standard deviation. When the fixed batching interval is too short (40 ms), the store is overloaded and we see large queuing delays. When the fixed interval is too long (at 70 ms and above), we see unnecessary latency. Figure 10 (d) shows a similar comparison, but with an offered load of 26.4k requests/second. Here we see



(a) Stout, 24k RPS (b) Stout, 26.4k RPS

(c) Fixed, 24k RPS (d) Fixed, 26.4k RPS

Figure 10: *Latency of responses for Stout (a, b) and fixed batching intervals (c, d), at two different workloads, 24k requests/second (a, c) and 26.4k requests/second (b, d). In (a, b), we see Stout's changing response latency overlayed with its mean response latency. In (c, d), Stout's mean response latency is overlayed with the mean latency and standard deviation for multiple fixed batching intervals. The slight increase in requests/second causes the best fixed interval from 24k requests/second to generate queuing at 26.4k requests/second.*

that the best fixed interval is at 60 ms, and the overhead imposed by Stout's adaptation is about 25 ms (from 75 to 100 ms), again within the standard deviation. If we use the best fixed interval from 24k requests/second (50 ms), the store becomes overloaded and unable to process requests in a timely fashion until the load subsides. These results demonstrate the need for adaptation—choosing the right fixed interval is difficult, even with this modest difference in offered load.

5.4 Dynamic Load Changes

Thus far we have shown Stout operating over fixed request rates. Here, we explore Stout's response to a sudden change in request load. For this experiment we apply a fixed load of 12k requests/second to our standard configuration and part way through the experiment we change the request load. Figure 11 shows the front-end latency for two of these experiments. In the first experiment, the load decreases to 6k requests/second. The front-end latency for Stout decreases from 50 ms to 5 ms. In the second experiment, the load increases to 18k requests/second and the latency increases from 50 ms to 80 ms. In contrast, a 20-ms fixed interval is marginally better than Stout at 12k requests/second but it only achieves 24 ms after the decrease and it causes queuing at the store after the increase. This demonstrates Stout's benefits in the presence of workload changes.

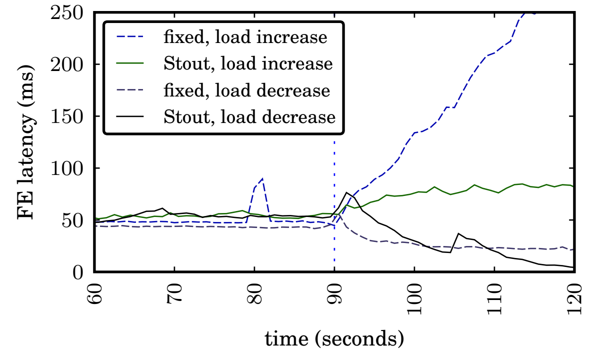


Figure 11: *Stout outperforms a fixed batching interval after the load either increases or decreases.*

5.5 Fairness

Cloud storage systems typically serve many middle-tiers and it is important that these middle-tiers obtain fair usage of the store. To measure Stout's ability to converge to fairness, we ran an experiment where after 90 seconds, we forcibly set half of the thirty-two middle-tiers to a batching interval of 400 ms and the remaining half to 80 ms. The middle-tier servers then collectively reconverge to the steady state. Because Centrifuge balances the distribution of documents across the middle-tiers, they have identical throughput throughout the experiment and we are only concerned with latency-fairness. The middle-tiers achieve good fairness after re-convergence: measuring from 30 seconds after the perturbation to 120 seconds after the perturbation, the mean latencies have a Jain's Fairness [22] of 0.97, where a value of 1.0 is optimal.

5.6 Alternate Storage Layers

To explore the generality of Stout's adaptation algorithm, we run experiments using two additional storage platforms with substantially different architectures. For both, we keep the same algorithm but calibrate the parameters to the new store. We first evaluate Stout against SQL Data Services (SDS) [30], a pre-release commercial storage system. For SDS, we calibrate the parameters to be the same as in Section 3.2 except that *thresh* = 0.2

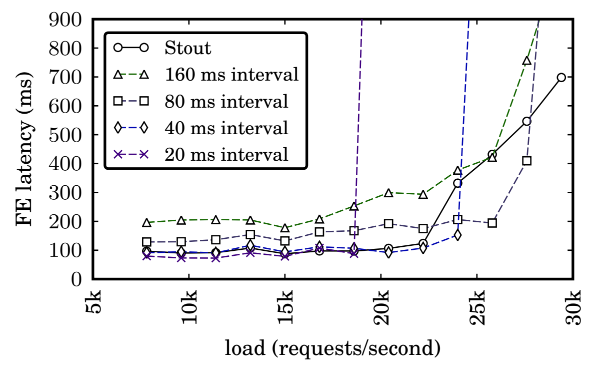


Figure 12: *Mean response latency for writes using PacificA: Stout and fixed intervals over a variety of loads.*

and $\beta = 1/4$. The current SDS API does not support batching or pipelining, and thus the best approach in our workloads is to send as rapidly as possible. We find that Stout does converge to sending as rapidly as possible.

We also evaluate Stout against PacificA [26], a research system that differs from our SQL-based storage layer in that it includes replication and uses log-based storage. We configure PacificA with three-way partitioning and three-way replication for a total of nine storage machines and one additional metadata server. The rest of the setup consists of twelve front-ends, sixteen middle-tiers, and one Centrifuge manager server. We calibrate the parameters from Section 3.2 to have EWMA-factor$= 1/32$, $thresh$ = 0.7, and $\beta = 1/8$. Figure 12 shows Stout's behavior across a range of request loads. At low to moderate load, Stout compares favorably to the best (20- and 40-ms) fixed batching intervals. As load increases, PacificA's log compaction frequency also increases, resulting in sufficiently frequent store hiccups that we are not able to avoid them in our experiments. After 22.2k requests/second, Stout has difficulty differentiating the store hiccups from the queuing behavior to which it is adapting. In spite of these hiccups, Stout outperforms any fixed batching interval in the presence of significant workload variation: compared to the short intervals, it avoids queuing at high loads; compared to long intervals, it yields much better latency at low loads.

5.7 Store Hiccups

As mentioned in our experiments with PacificA, stores sometimes experience hiccups, where they briefly pause in processing new requests. Such Stout-independent hiccups can lead to large spikes in observed latency, complicating Stout's task of inferring store load. We now investigate the issue of hiccups in more detail.

Figure 13 shows the occasional brief pauses in processing (or "hiccups") that occur over a 2-hour interval when using the SQL Server storage system. For this experiment, we used a single middle-tier server sending 3k operations per second with a fixed 2-ms batching interval to a single SQL Server back-end machine, and we

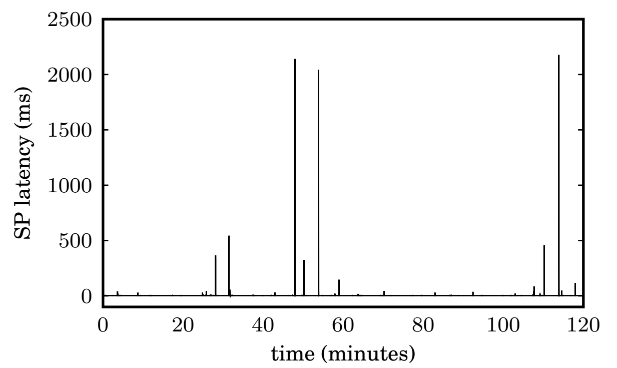


Figure 13: *Intermittent hiccups in store processing yield brief spikes in latency as measured from the middle tier. These measurements were taken with a 2-ms fixed batching interval and 3k requests/second.*

measured latency from within the Stout storage proxy — this is denoted SP latency and it only includes the time to send the requests over a TCP connection to the back-end and the time that the store takes to service these requests and send responses back to the middle-tier. The figure shows that these hiccups occur on an irregular and infrequent basis, and they lead to significant spikes in latency — up to three orders of magnitude greater than the steady state. Although this figure only shows the hiccups at one offered load, we have run similar experiments with different loads, and we have not observed any obvious correlation between the offered load and the frequency of hiccups in this store.

Although we do not know the exact cause of hiccups in the SQL store, we believe they are caused by periodic background bookkeeping tasks that are common in storage systems. We did make efforts to eliminate such hiccups from SQL Server by both disabling the option that generates query-planning statistics and setting the recovery interval to one hour (the recovery interval controls how much replay from the log may be needed after a crash). These changes reduced the number of hiccups but did not eliminate them. As mentioned in Section 5.6, we observed that log compaction is responsible for even more frequent hiccups in PacificA.

Because these brief latency spikes may be unrelated to the offered load, an appropriate response to them is simply to pause briefly; increasing the batching interval is not appropriate because the store is not actually overloaded. The problem of a unrelated event causing the appearance of congestion is familiar from the literature on TCP over wireless channels, where packet loss may reflect either congestion (which should be mitigated by the sender) or background channel noise (which can frequently be ignored). In response, researchers have proposed explicit signaling techniques like ECN [4, 25] to improve performance in these challenging environments. Our measurements suggest that similar mechanisms for

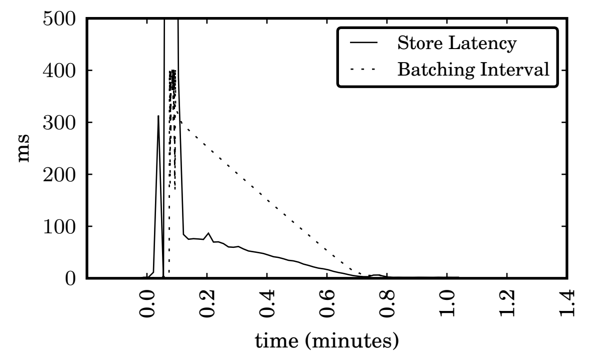


Figure 14: *Stout recovering from a store hiccup while operating at 3k requests/second.*

adaptive use of cloud storage are also worth researching. In this paper, we restrict our attention to showing that Stout, which does not try to distinguish latency due to store hiccups from latency due to overload, still copes acceptably with such hiccups.

Figure 14 shows how Stout reacts to one of these hiccups: the solid line shows the measured response time of the store, and the dashed line shows how Stout adjusts its $intrvl$ as a result of the latency spike. With $intrvl_{max}$ set to 400 ms, Stout takes slightly over half a minute to recover from the very large spike in latency (the peak in this figure is 2,696 ms) caused by this hiccup. This recovery is rapid compared to the frequency of hiccups. Lowering $intrvl_{max}$would improve recovery time, but would also reduce Stout's operating range.

The rarity of store hiccups raises a methodological question: each of our experiments would have needed to run for hours in order for the number of hiccups to be similar across runs. Because Figure 8 alone includes 27 such experiments, such an approach would have significantly hindered our ability to evaluate Stout under a wide variety of conditions. Because Stout recovers from store hiccups with reasonable speed, we chose instead to re-run the occasional experiment that saw such a hiccup. The one exception is our experiment using PacificA (Section 5.6), where hiccups were sufficiently frequent that we did not need to take any special steps to ensure a comparable number across runs.

6. RELATED WORK

Stout's control loop is inspired by the literature on TCP and, more generally, adaptive control in computer systems. The Stout implementation also incorporates a number of well-known techniques from storage systems. We briefly discuss a representative set of this related work.

There is a large existing literature on TCP [21, 24, 43]. This prior work has explored many different indicators of utilization and load; Stout uses response time measurements to adjust its rate of sending requests to the store. In this regard, Stout is similar to TCP Vegas [5], FAST TCP [41] and Compound TCP (CTCP) [37], each of which attempts to tune the transmit rate of a TCP flow based upon the inter-packet delay intervals. In comparison, Stout's control loop has to deal with the additional subtlety of distinguishing delay due to congestion from delay due to sending a larger batch.

Control theory is a deep field with many applications to computer systems [42, 38, 8, 34, 28, 9]. Despite these successes, many adaptation problems in computer systems have remained unaddressable by control theory due to the dramatic differences between computer systems and the systems that control theory has traditionally considered [18]. For example, advocates of a class of controllers called self-tuning regulators have constructed a list of eight requirements that computer systems must satisfy to enable their successful application [23]. Scale-out storage systems fail to satisfy a number of these conditions, such as the requirement for a modest bound on the actuation delay of the system (e.g., if an application enqueues a large number of requests, future request batching can take a very long time to reduce user-perceived latency). Other control techniques may remove this particular requirement, but instead introduce other difficult requirements, such as the need for a detailed model of scale-out storage system performance [23].

The Stout implementation borrows from prior work on storage systems in two major ways. First, the performance benefits of batching, write collapsing and pipelining are well-known, and have been leveraged by systems such as Lightweight Recoverable Virtual Memory (LRVM) [36], Low-Bandwidth File System [32], Farsite [1], Cedar [16], Practical BFT [6], Tandem's B30 system [17] and the buffer cache [40]. Stout's novelty is in using a control loop to manage exploiting these optimizations, not the optimizations themselves.

Second, Stout's internal architecture incorporates at least two major ideas from prior storage systems. Splitting consistency management from storage was explored in Frangipani [39] and LRVM [36], while prior work such as Soft Updates [14], Generalized File System Dependencies [13], and xsyncfs [33] explored ways to provide some or all of the performance benefits of delayed writes with better consistency guarantees.

7. CONCLUSION

Stout's adaptation algorithm is the first technique for automatically adapting application usage of scalable key-value storage systems. Stout treats store access as a congestion control problem, measuring the application-perceived latency and throughput of the store, and dynamically adjusting the application's grouping of requests to the store. To evaluate this algorithm, we implemented the Stout system and modified a real-world cloud service to use Stout. We found that in the presence of significant workload variation, Stout dramatically outperforms non-adaptive approaches.

Acknowledgements

We thank the anonymous reviewers, and our shepherd, Garth Gibson, for their feedback on earlier drafts of this manuscript.

8. REFERENCES

[1] A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, J. Howell, and J. Lorch. Load management in a large-scale decentralized file system. Technical Report MSR-TR-2004-60, Microsoft Research, July 2004.

[2] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. In *Proceedings of USENIX NSDI*, Apr. 2010.

[3] Azure Storage. `http://www.microsoft.com/azure/windowsazure.mspx`.

[4] H. Balakrishnan and R. Katz. Explicit Loss Notification and Wireless Web Performance. In *Proceedings of the IEEE Globecom Internet Mini-Conference*, 1998.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM*, pages 24–35, Aug. 1994.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX OSDI*, 1999.

[7] 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 *Proceedings of USENIX OSDI*, Nov. 2006.

[8] J. S. Chase, D. C. Andersen, P. N. Thakar, A. Vahdat, and R. P. Doyle. Managing energy and server resources in hosting centres. In *Proceedings of ACM SOSP*, pages 103–116, Oct. 2001.

[9] C. M. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proceedings of VLDB*, pages 342–353, Aug. 1993.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of USENIX OSDI*, Dec. 2004.

[11] 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 *Proceedings of ACM SOSP*, pages 205–220, Oct. 2007.

[12] Disable SATA Write Caching. `http://support.microsoft.com/kb/811392`.

[13] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In *Proceedings of ACM SOSP*, pages 307–320, Oct. 2007.

[14] G. R. Ganger and Y. N. Patt. Metadata update performance in file systems. In *Proceedings of USENIX OSDI*, pages 49–60, Nov. 1994.

[15] Google. Google Apps: Gmail, Calendar, Docs and more. `http://apps.google.com`.

[16] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. *SIGOPS Operating Systems Review*, 21(5):155–162, 1987.

[17] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group Commit Timers and High Volume Transaction Systems. In *Proceedings of High Performance Transaction Systems*, pages 301–329, 1989.

[18] Y.-C. Ho. On centralized optimal control. *IEEE Transactins on Automatic Control*, 50(4):537–538, 2005.

[19] Hotmail. `http://www.hotmail.com`.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of ACM EuroSys*, Mar. 2007.

[21] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM*, pages 314–329, Aug. 1988.

[22] R. Jain, D. M. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, Digital Equipment Corp., Sept. 1984.

[23] C. Karamanolis, M. Karlsson, and X. Zhu. Designing controllable computer systems. In *Proceedings of USENIX HOTOS*, 2005.

[24] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communications Review*, 33(2):83–91, 2003.

[25] A. Kuzmanovic. The Power of Explicit Congestion Notification. In *Proceedings of ACM SIGCOMM*, pages 61–72, Aug. 2005.

[26] W. Lin, M. Yang, L. Zhang, and L. Zhou. PacificA: Replication in log-based distributed storage systems. Technical Report MSR-TR-2008-25, Microsoft Research, 2008.

[27] Live Mesh. `http://www.mesh.com`.

[28] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of ACM SOSP*, pages 238–251, Oct. 1997.

[29] Microsoft. Office Web Applications. `http://www.microsoft.com/Presspass/Features/2008/oct08/10-28PDCOffice.mspx`.

[30] Microsoft. SQL Data Services. `http://www.microsoft.com/azure/data.mspx`.

[31] M. Moshayedi and P. Wilkison. Enterprise SSDs. *ACM Queue*, 2008.

[32] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187. ACM New York, NY, USA, 2001.

[33] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of USENIX OSDI*, pages 1–14, Nov. 2006.

[34] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of ACM EuroSys*, pages 289–302, Mar. 2007.

[35] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[36] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):146–160, Feb. 1994.

[37] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound TCP approach for high-speed and long distance networks. In *Proceedings of IEEE Infocom*, pages 1–12, Apr. 2006.

[38] C. Tang, S. Tara, R. Chang, and C. Zhang. Black-Box Performance Control for High-Volume Non-Interactive Systems. In *Proceedings of USENIX ATC*, June 2009.

[39] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of ACM SOSP*, pages 224–237, Oct. 1997.

[40] W. Vogels. File system usage in Windows NT 4.0. In *Proceedings of ACM SOSP*, pages 93–109, Dec. 1999.

[41] D. Wei, C. Jin, S. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (TON)*, 14(6):1246–1259, 2006.

[42] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of ACM SOSP*, pages 230–243, Oct. 2001.

[43] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *Proceedings of INFOCOM*, Mar. 2004.

IsoStack – Highly Efficient Network Processing on Dedicated Cores

Leah Shalev, Julian Satran, Eran Borovik, Muli Ben-Yehuda
leah@il.ibm.com, Julian_Satran@il.ibm.com, borove@il.ibm.com, muli@il.ibm.com

IBM Research – Haifa

Abstract

Sharing data between the processors becomes increasingly expensive as the number of cores in a system grows. In particular, the network processing overhead on larger systems can reach tens of thousands of CPU cycles per TCP packet, for just hundreds of "useful" instructions. Most of these cycles are spent waiting – when the CPU is stalled while accessing "bouncing" cache lines of network control data shared by all processors in the system – and synchronizing access to this shared state. In many cases, the resulting excessive CPU utilization limits the overall system performance. We describe an IsoStack architecture which eliminates the unnecessary sharing of network control state at all stack layers, from the low-level device access, through the transport protocol, to the socket interface layer. The IsoStack "offloads" network stack processing to a dedicated processor core; multiple applications running on the rest of the cores invoke the IsoStack services in parallel, using a thin access layer that emulates the standard sockets API, without introducing new dependencies between the processors. We present a prototype implementation of this architecture, and provide detailed performance analysis. We demonstrate the ability to scale up the number of application threads and scale down the size of messages. In particular, we show an order of magnitude performance improvement for short messages, reaching the 10Gb/s line speed at 40% CPU utilization even for 64 byte messages, while the unmodified system is choked when driving 11 times less throughput.

1. Introduction

While networking demands in data centers continue to grow, and the networking infrastructure continues to provide improved bandwidth and latency, single processor performance remains the same and in some cases even decreases. Recently, increasing the number of CPU cores became the only way to perform more instructions per cycle. However, the overhead due to interaction between these cores also goes up, and naïve data-sharing may inhibit performance scaling as the number of cores grows. Nevertheless, the familiar shared memory programming model is still commonly used for both application programming and implementation of OS services.

Since the days of uniprocessor systems, network processing has been carried out in a "multithreaded" fashion: some portions of the stack are executed during the socket system calls (in the context of calling applications), others during receive packet processing (in the context of interrupt handlers or kernel threads owned by the network stack), and yet others in the context of timeout handler routines. As multiprocessors were introduced, it was natural to distribute these stack processing elements symmetrically on the multiple processors in order to keep pace with the growing networking speeds. As the number of processors grows, the cost of sharing the network control structures between the processors becomes extremely high; meanwhile, cores become so abundant that sparing a few becomes feasible. This has provided an opportunity to re-think the network stack architecture and take advantage of the changing landscape of computer systems.

The IsoStack is a different approach for integrating network processing within a multicore system. Instead of using the cores symmetrically, the IsoStack uses dedicated cores for network processing, and leaves the rest of the cores for running applications. Since the network processing is confined to dedicated processors, the stack can be optimized – executed serially without interrupts and locks. Since the CPUs are not shared between applications and the stack, there are fewer context switches, and the cache behavior is improved. The IsoStack provides applications with a high-level interface (similar to a TCP Offload Engine interface), which can also allow efficient virtualization support using simple HW devices.

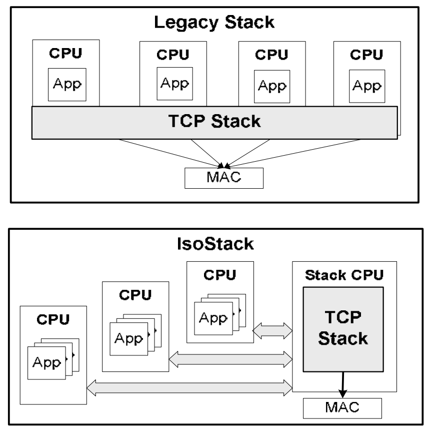


Figure 1. Native stack vs. IsoStack

The contributions of this paper are:

- The architecture of an isolated network stack that allows independent, contention-free, execution of TCP/IP control operations on a dedicated core, and application data processing on the other cores;
- The prototype implementation of such a stack in AIX 6.1 on Power6, providing a standard synchronous socket API built upon an asynchronous internal interconnect;
- Implementation of an optimized message queue mechanism for internal communication between a large number of applications (producers) and a consumer running on a dedicated core;
- The performance evaluation for a 10 Gb/s link, demonstrating a significant increase of bandwidth and/or decrease of total CPU utilization compared to the native stack, in some cases yielding an order-of-magnitude improvement.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 describes the system architecture, and Section 4 depicts the prototype implementation. We present the experimental results in Section 5, and conclude the paper in Section 6.

2. Background and Related Work

For decades, TCP performance optimizations were introduced gradually to address the performance hot spots of contemporary systems ([1, 2, 5]). The most widely adopted optimizations include checksum calculation offload, interrupt mitigation to decrease the number of interrupt requests from networking devices, and techniques that decrease the number of packets to be processed for bulk data transfer. Some of these techniques for decreasing the number of packets include jumbo frames ([5]), Large Send Offload (LSO [31]), also called TCP segmentation offload (TSO), and, recently, Large Receive Offload (LRO [21, 25]). Nevertheless, the resulting improvements merely succeeded to compensate for the rapidly growing networking demands, combined with relatively slow growth of CPU speed and even slower improvement of memory bandwidth and latency ([6, 16]).

With the advent of multiprocessor and (later) multicore systems, stack parallelization became necessary to keep pace with the growing network bandwidth. However, efficient parallelization remains challenging, as the parallel stack architectures implemented in the modern operating systems incur additional locking overhead, cache inefficiencies, and scheduling overhead ([23]).

Receive-Side Scaling ([18]) and similar techniques let a NIC classify the incoming packets to determine the affinity between these packets and CPU cores. On the basis of the packet classification result, received packets are dispatched to the appropriate receive queue, which is usually served by a particular processor. This technique allows more efficient low-level device sharing, as it relieves the bottleneck associated with sharing a single receive queue, and instead allows the stack to process received packets in truly parallel way when the packets are independent (i.e., belong to different sets of network flows). On special-purpose systems (such as embedded network appliances), running customized applications, this could potentially allow to confine all TCP processing for a particular connection to a single processor core. However, on general-purpose systems (running regular sockets applications), if the rest of the sharing issues are not addressed, RSS (as well as other receive-side optimizations such as NAPI) only allow to eliminate a small part of the multiprocessing overhead. This is because the receive processing, the transmit processing and the timer processing for the same TCP connection are still likely to be executed on different processors. In particular, application-triggered data transmission is executed in application thread context, while ACK handling and ACK-triggered data transmission are executed by the receive handler. The transmit thread either does not have any CPU affinity, or its affinity is configured by the application, while the affinity of the receive handler is configured by the operating system, transparently to the application. Also, an application thread can handle multiple connections, that can be mapped by RSS to different CPUs. Accordingly, such un-coordinated execution still necessitates locking to protect access to the TCP connection and the associated socket state, and may cause cache line bouncing when accessing this state.

A radical solution to the fast-network, slow-host phenomenon is offered by RDMA approach ([10]). It offloads the protocol to an RDMA-enabled adapter, which allows zero-copy operation due to RDMA semantics, and eliminates per-packet overhead due to offloaded transport processing. Although this approach is suitable for high-performance computing applications running in a closed environment and using MPI or explicit RDMA semantics API, it is not feasible for data-center applications using sockets API, implementing standard protocols (such as HTTP) directly over TCP, and interacting with legacy clients. For this latter class of applications, pure TCP offload (without RDMA semantics) has been proposed.

TCP offload for socket applications has been pursued for a long time ([8, 11, 12, 13, 19]), and remains controversial. Its potential advantage is the improved performance due to a higher-level interface that decreases the amount of interaction between the software and the TCP Offload Engine (TOE) adapter, since the internal events are handled by the TOE adapter and do not disrupt application execution. However, in practice, the performance potential of TOE materializes only under various limitations. For example, it may be necessary to modify the existing applications in order to achieve improved performance. Also, due to high complexity and low volumes, TOE solutions tend to have high cost and longer development cycle comparing to the rest of the system components, which can make a TOE engine obsolete by the time it is released. In addition, TOE solutions lack the flexibility in protocol processing that is needed to support future protocol changes, and are prone to bugs that cannot be easily fixed. Even if the internal implementation is programmable, the changes can only be done by the adapter vendor, leaving the OS very little control over the protocol behavior. This impedes TOE support in some operating systems, and hinders TOE acceptance in general.

“TCP onload” using a dedicated CPU was proposed for multiprocessor systems as an alternative to TCP offload, without the disadvantages of hardware-based TCP offload ([14, 15, 17, 20]). The concept is based on an asymmetric multi-processing mode, where at least one of the CPUs on a multiprocessor system is dedicated to network stack processing, serving as an integrated TCP offload engine. This architecture allows a significant reduction in overhead when compared to naïve parallelization approaches. The TCP Servers project ([7]) also demonstrates the value of a similar approach. However, the previous solutions for CPU-based TCP offload made simplifying assumptions on the interaction between the applications and the onloaded stack, and did not demonstrate performance improvement for inconveniently small message sizes or for high number of applications sharing the "onloaded" services. The IsoStack work is focused on improving these aspects of the onload concept.

Loosely coupled TCP acceleration ([22]) is a hybrid approach that combines the benefits of both offload and onload. Similar to the offload approach, the application CPU uses a lightweight interface to interact with an “offloaded” network stack. However, network stack processing is not fully offloaded to the network interface adapter. Instead, only the data processing is performed by a hardware acceleration engine on the adapter, while the protocol control operations are done by software on a dedicated main CPU. The software and hardware components are loosely coupled; the parallelization is done in a way that allows asynchronous and independent operation of both parts. In particular, the control information that has to be accessed by both entities is replicated rather than shared, using message queues to explicitly exchange state changes.

The same principle of dividing up responsibilities was also applied in the Scalable I/O project ([26]), which showed that efficient and scalable I/O virtualization becomes possible by complete separation of the I/O and compute functions. Moreover, the OS structure itself can be revisited to reduce unnecessary sharing, as in the Corey operating system for many cores ([27]); or to eliminate the sharing altogether, as in the Multikernel architecture ([28]). Asymmetrical OS structure was also employed in the Piglet operating system ([4]) which used dedicated processors to implement "intelligent device" functions.

3. IsoStack Architecture

In this section we present the IsoStack architecture, in which we confine the network protocol processing to dedicated processors and isolate it from the application execution environment.

The IsoStack architecture is guided by the following design principles:

- Serialized, event-driven, lock-free, and interrupt-free implementation of the IsoStack on one or more dedicated logical processors. In particular, adapter control structures are not shared between processors.
- Asynchronous interaction between applications and the IsoStack, through explicit messaging, without the sharing of state.
- The isolation is transparent to applications; in particular, the underlying asynchronous protocol does not affect the latency of synchronous operations.

The first two design principles allow more efficient implementation of the network stack, with better utilization of multiple processors. This is due to elimination of the overhead caused by access to shared data structures from different processors and better use of each processor's resources (e.g., decreased cache pollution). The last principle allows unmodified applications to benefit from the improved stack performance, without having to switch to a different API or make any other changes.

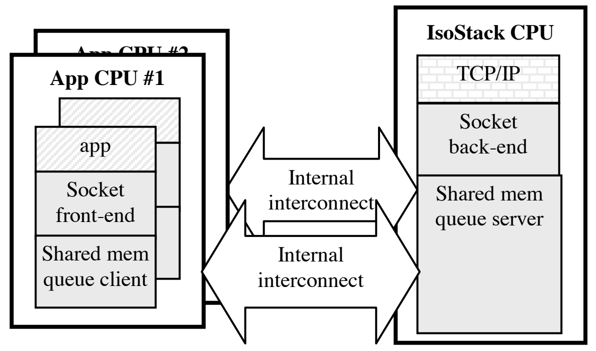


Figure 2. IsoStack architecture

The IsoStack architecture is depicted in Figure 2. Applications access network services using a socket front-end layer that implements the standard socket API and replaces the legacy sockets layer. The socket front-end handles the API peculiarities and delegates the execution of networking operations to the socket back-end. The socket front-end and the socket back-end interact using an asynchronous protocol over an internal interconnect. The architecture allows different types of internal interconnects. In our earlier work ([26]), we used Infiniband ([9]) for communication between the socket front-end and back-end. This work focuses on a message queue mechanism using the available general-purpose hardware; namely, cache-coherent memory; the detailed discussion is in Section 4.1.

Socket back-end receives network commands from socket front-end, executes the commands asynchronously and sends the command status in the opposite direction. The commands include socket transmit/receive/control commands, and buffer registration commands. Different APIs, such as standard synchronous BSD sockets or various flavors of asynchronous sockets, can be implemented using the same underlying command/status mechanism. For example, the asynchronous Extended Sockets API ([33]), which exposes explicit memory registration of application buffers, allows transmit implementation with true zero-copy. The standard socket API can be implemented with a single data copy into the socket transmit buffer, using in-advance registration of that internal socket buffer, as described in Section 4.2.

The IsoStack uses a dedicated logical CPU, and is solely responsible for all network processing for a particular network interface, which eliminates contention on access to network control data structures and allows a wide range of optimizations. Since the processor is not shared with other components, context switching overhead is reduced, and polling-mode interrupt-free execution becomes possible, eliminating the interrupt handler overhead. Since the data structures are not shared with other processors, single-threaded, serialized execution enables lock-free operation, thus eliminating the locking overhead. Consequently, all major sources of stack inefficiency are removed.

Although this paper focuses on the case of a single IsoStack processor and a single network interface assigned to it, this is not an architectural limitation. It is possible to run multiple independent IsoStack instances, where each IsoStack instance is responsible for one or more network interfaces. Moreover, since hardware support for packet classification (with multiple receive queues) is common, throughput scaling for a single network interface can be achieved by using several independent instances of the IsoStack, each responsible for a subset of network traffic flows on that interface, as discussed in Section 6.

On the other hand, it is not necessary to consume completely a processor core under light load. In order to save power when the traffic rate is low, the IsoStack can temporarily enable the interrupts and stop the polling until it is notified on a new event. The interrupt handlers in this case are used only to resume the polling, hence this type of interrupt-driven execution does not re-introduce the shortcomings of the regular stack implementation.

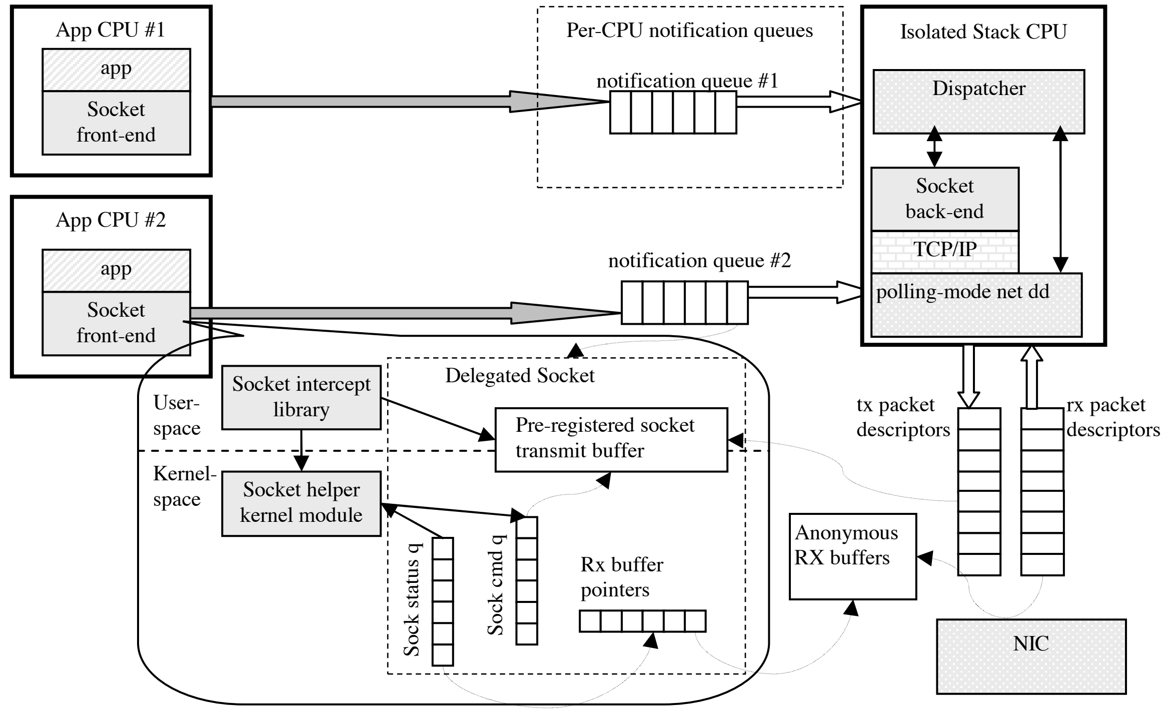


Figure 3. System Design

4. Prototype Implementation

The IsoStack prototype is based on the AIX 6.1 operating system, running on a Power6 system using the HEA 10Gb/s adapter. We modified several kernel components to allow the isolated-mode operation of the network stack as a single kernel thread, added new kernel extension modules to support "delegation" of socket operations to the IsoStack, and implemented a user-space library that intercepts socket operations and passes them to the IsoStack instead of invoking the socket system calls. Figure 3 depicts the high-level system design.

The socket layer is split into socket front-end and socket back-end to accomplish the delegation of socket operations. In particular, the state of each socket is split into its socket delegation state at the front-end, while the actual socket object (including the network protocol control information) is maintained at the socket back-end. The socket front-end consists of a socket intercept library that primarily provides user-space implementation of standard socket calls, and a socket helper kernel module that facilitates communication between the socket front-end and back-end when kernel-level privileges are required (for example, to access shared notification queues, as explained in Section 4.1). The socket back-end is a part of the IsoStack; it receives socket commands from the socket front-end, and executes them using the asynchronous in-kernel socket APIs adapted for single-thread, interrupt-free operation.

Section 4.1 describes the design of the messaging mechanism used for the interaction between the socket front-end and back-end. Sections 4.2 and 4.3 provide details of the transmit and receive operations, respectively. Section 4.4 describes the event-driven operation of the IsoStack. Section 4.5 lists the lock elimination optimizations enabled by our architecture.

4.1 Message Queues

An efficient mechanism for interaction between the application and the IsoStack is critical for realizing the performance improvement potential of our architecture. Clearly, executing the network processing on a separate CPU, without the overhead of locks or interrupts, reduces the stack overhead. However, the separation introduces a new overhead, which must be kept very low in order to make the overall solution worthwhile. In particular, this necessitates a highly efficient many-to-one producer-consumer mechanism, to pass commands to the IsoStack from multiple applications.

The design of such a mechanism was one of the main challenges of this work. Our early experiments showed that the existing IPC services are too expensive in terms of both CPU utilization and latency. On the other hand, the existing solutions for lock-free, producer-consumer interaction via shared memory provide much better performance for low numbers of producers, but do not scale well as the number of producers grows, because the consumer must poll large numbers of queues. Ideally, a simple hardware mechanism could be employed to safely serialize request submissions from multiple non-cooperative, non-trusted clients to a single request queue, which could then be polled by the server. Such a mechanism could allow lock-free direct access to the queue by multiple producers, with atomicity handled by the hardware. Unfortunately, such a mechanism is not yet available, which makes the single queue approach unfeasible. The access to such a single queue becomes very expensive under the heavy contention due to the queue sharing by all socket applications (and all processor cores) in the system.

To decrease the cost of queue sharing, we chose to use a separate queue per logical processor (processor core or thread if SMT is in use). Thus, the number of queues is constant and small enough to allow efficient polling by the consumer. Each thread accesses (atomically) the queue of the processor on which it is running at the time of the access; the queue is not shared by other processors in the system, which allows contention-free producer operation. Unfortunately, since these queues are shared by different applications, they cannot be accessed directly from user-space; kernel-space socket helper provides protected access to the notification queues.

The per-CPU queues are used to notify the IsoStack of new application requests; the notification queue entries include only the socket identification information. The actual socket commands are kept in per-socket command queues that reside in shared memory, accessible to both socket front-end and socket back-end; the command responses are returned through per-socket status queues. The queues are implemented using the coherent shared memory in a controlled way, where each side maintains its view of the protocol state; all memory locations used to exchange information between the sides are allowed to be updated by a single designated writer (i.e., each shared memory location can be written by either the socket back-end or the socket-front-end within the appropriate application). Each application uses separate shared memory segment for writeable and readable parts of the queue state. Also, complete separation is maintained between the applications.

The design is somewhat similar to direct-access TCP offload solutions with interface comparable to Virtual Interface Architecture (VIA [3]), when the notification queues serve to emulate doorbells, and command/response queues are implemented as lock-free producer-consumer queues.

4.2 Socket Send Operation

One of the key issues in the design of efficient data transfer (for any type of I/O) is memory management. This issue is particularly complicated for communication services based on legacy, streaming-mode, synchronous socket API, due to inherent data copy semantics and unpredictable patterns of application operation. In particular, a large data transfer is likely to be implemented as a sequence of multiple smaller transfers, invoked synchronously, passing data residing at arbitrary locations. This observation, together with the fact that the data copying overhead becomes less pronounced on modern systems ([24]), underlies our decision to avoid zero-copy design for socket send operations – even though such a decision seems counter-intuitive, as zero-copy property is considered a holy-grail of network acceleration solutions. Zero-copy solutions tend to offer improved performance at the cost of application modification (e.g., through new asynchronous APIs), and are only beneficial for a subset of workloads. We, on the other hand, strive to improve performance for a broad range of existing unmodified applications. In particular, one of our design goals was to keep (or improve) the low latency of the synchronous send call. Thus, we chose to keep the single data copy, performed on the application side.

In our solution, the synchronous API is implemented using socket transmit buffers that are pre-allocated and pre-registered for the DMA access. This significantly reduces buffer management overhead and allows efficient aggregation of small data chunks. The socket back-end allocates DMA-able memory segments for each socket application; during socket initialization, the socket front-end (kernel helper) allocates per-socket transmit buffers out of the DMA-able chunk and maps them for user-space access. When the application sends data, the socket front-end copies the data from the application buffers into the socket transmit buffer (mapped into the application address space) used as a contiguous cyclic buffer. Afterwards, the socket front-end writes a transmit command to the socket command queue, specifying the location of new transmit data within the socket buffer. To simplify memory protection, it does not use pointers to identify the data in the transmit buffer, and instead uses offsets relative to the buffer start. When the socket back-end receives the command, it uses the buffer registration information and the specified offset to construct the DMA address to be passed to the device driver. The socket back-end does not access the transmit buffers; it just serves as an intermediary that facilitates the buffer sharing between the socket front-end and the NIC.

The implementation of the send call copies the application data to the transmit buffer; the space occupied by the copied data is reused after the socket back-end reports that it was delivered to the remote receiver. The buffer space is used to facilitate the batching of multiple small requests in case the sender is faster than the local stack or the receiver. The socket front-end does not necessarily notify the socket back-end about each new piece of data that was copied to the transmit buffer. Instead, it aggregates data if the amount of previously posted pending data becomes high, until the socket back-end reports sufficient progress on the data transmission, or until a large amount of data has accumulated. Thus, the data aggregation does not increase latency; it occurs only when the previously submitted data starts piling up.

In turn, the socket back-end performs additional aggregation, postponing the TCP processing of newly submitted data when the TCP connection state does not allow immediate segment generation (i.e., when the TCP send window or congestion window is full). Like the aggregation at the socket front-end, the aggregation at the socket back-end does not introduce unnecessary delays; it decreases the TCP overhead and the overhead of the interaction with the device, due to better utilization of its TCP segmentation capabilities.

4.3 Socket Receive Operation

Handling incoming network traffic using a regular NIC is a known challenge. Due to unpredictable patterns of packet arrival, the packets received by a stateless NIC must land into anonymous buffers that are not associated with a particular connection. The packet data must be copied from the anonymous kernel buffers to the application buffers, which may be provided by the application after an arbitrary delay; thus, complex bookkeeping of the packet data structures is needed. The main design choice we had to make was the context for performing the data copy operation.

One choice would be asynchronous copy by the socket back-end, which seems to offload a maximal number of CPU cycles from the application CPUs. However, this approach has numerous drawbacks. It causes thrashing of the IsoStack resources such as cache, TLB, and SLB, and it may actually decrease the application performance due to increased latency of receive operation and decreased cache locality; this occurs when the application tries to access the newly received data, which was brought to the wrong cache during the copy. Accordingly, we decided to copy the data on the application CPU, within the socket front-end.

Applications (or their writers) expect the latency of the receive socket call to be very low if the data already arrived. In order to minimize this latency, our implementation strives to perform the copy during the synchronous execution of the receive call, without interacting with the socket back-end. To achieve that, the socket front-end "prefetches" receive buffers from the socket back-end in advance, independently of the receive calls invoked by the application, using asynchronous requests. Upon such request, the socket back-end hands over to the socket front-end the ownership on the data buffers that contain the receive data stream of the socket (when these are available). Multiple buffers corresponding to multiple network data segments can be reported at once, decreasing the interaction between the socket front-end and the back-end. If the previously posted request is completed before the application invokes socket receive function, socket receive implementation in the socket front-end copies the data immediately; otherwise, the application blocks until the previously requested data buffers are available. The socket front-end uses a heuristic to decide when to request more buffers.

Since the packet buffers reside in kernel space and cannot be mapped in advance to the relevant application, the receive pointers queue is maintained in the kernel by the socket helper kernel module, which also copies the data during the socket receive call invoked by the application. This necessitates a kernel boundary crossing upon each receive operation, thus incurring a higher overhead than the send. However, the overhead is still lower than the native implementation because the socket front-end state is only accessed locally, unlike the regular socket object in the native stack, which is shared between different stack components running in different contexts.

4.4 Event-Driven IsoStack Operation

The IsoStack is implemented as a single-threaded non-preemptive processing loop, serially handling asynchronous events. A dispatcher component of the IsoStack polls event queues to detect the new work to be done such as new packet arrivals, new application requests to be executed, or timeout expiration; it then invokes appropriate event handlers sequentially. The device is configured to operate in polling mode; a new device driver entry point is used to poll periodically for new packet arrival. The message queue mechanism also allows periodic polling of the socket command queues (or, more precisely, event notifications queues). The polling is done by reading from a cache-coherent memory location, thus busy-wait polling on empty queues is inexpensive, because it is usually accomplished by access to the local cache only.

The socket back-end running within the IsoStack executes the commands delegated by the socket front-end. If it cannot execute a command immediately, it postpones the command execution until an appropriate change of state occurs (e.g., until incoming data is buffered, in the case of the receive command). Each such command is implemented as a separate state machine. For example, if the socket front-end is requested to send data on a socket when the transmit window is full, the command handler puts aside the command state and marks the socket to enable asynchronous notification when transmission becomes possible. It then returns, allowing the dispatcher to proceed with other work. When an ACK packet arrives on the appropriate connection, the adapter's polling receive handler (invoked by the dispatcher) passes the packet up the stack; the TCP processing layer performs its regular processing and then generates an internal event indicating that the window space is freed. Later, the dispatcher detects the internal event and passes it to the socket back-end, which resumes execution of the send command.

4.5 Lock Elimination

Our architecture allows elimination of locks that were introduced within the network stack as a part of support for multiprocessor systems. Since the socket back-end objects and the network interface data structures are accessed sequentially in the context of the IsoStack thread, there is no need to worry about mutually exclusive access for these resources, which are private to the IsoStack. We made minimal modifications to the appropriate stack components to bypass the locking/unlocking code when touching the device or socket resources that belong exclusively to the IsoStack.

Many other stack resources, such as the hash table of TCP connections or IP routing table, are shared across the system. To better utilize the advantages of our architecture, it is desirable to avoid this sharing and allow local-only access instead. These structures can be split into independent instances, each holding the relevant portion of information, potentially replicated and updated only using explicit "messages" delivered as internal events. For example, the generic Ethernet handling layer uses a lock to protect access to shared device configuration information that is changed rarely, if ever, using management interfaces. In our architecture, the IsoStack must be the exclusive owner of configuration information for the devices assigned to it; the management interfaces need to be intercepted, and execution of configuration changes need to be delegated to the IsoStack. This would make locking unnecessary, since the device configuration is accessed serially. Our experiments show that even uncontended locks incur a high overhead; thus, elimination of these remaining locks can yield an additional tangible performance improvement.

5. Experimental Results

This section demonstrates the performance improvement that can be achieved using the IsoStack approach. We use several micro-benchmarks to emulate different workloads, and evaluate the performance of several variants of the IsoStack, using the native (unmodified) stack as a baseline.

5.1 Experimental Setup

Our system under test is a Power6 machine, connected back-to-back to a "remote" system over a 10Gb/s link. Both machines have an additional NIC used for remote access. The Power6 system is a 4-way (8 core) system, running at 3.5 GHz, with 16 GB of RAM, equipped with a 10Gb/s HEA (Host Ethernet Adapter). All physical resources are assigned to a single logical partition (LPAR), which runs the AIX 6.1 operating system. Since the cores provide two-way SMT (symmetrical multithreading) capabilities, the machine appears to have 16 logical processors from the point of view of the OS. The remote system is a quad core AMD Opteron machine with 2GB RAM, equipped with 10G Broadcom NetXtreme II BCM57710 NIC, running Red Hat Enterprise Linux 5.3 (2.6.18 kernel).

Our experiments compare the AIX native TCP/IP stack with the IsoStack, using the same micro-benchmark applications. To measure the IsoStack performance, we ran the IsoStack socket back-end and the test applications linked with the socket front-end. To obtain AIX native results, we re-ran the same tests linked with the regular socket library over the unmodified AIX kernel and the unmodified network drivers with the

same adapter configuration parameters. To achieve maximum bandwidth (on both types of systems), we increased the dedicated interfaces' MTU to 9000, disabled hardware flow control, and enabled TCP checksum offload and TCP segmentation offload. The AIX built-in Nmon tool ([32]) was used to measure network throughput and CPU utilization.

In order to evaluate scalability of our implementation for multiple application threads, we used a multi-threaded TTCP-like application, where each thread sends or receives data over a single socket. We measured the achieved throughput, and the total CPU utilization for all processors (i.e., 100% means all cores are fully utilized; a single core accounts for 12.5%). Note that the IsoStack core is always fully consumed, because of polling-mode operation. CPU utilization of IsoStack shown below includes the constant utilization of the IsoStack core, and varying CPU utilization of the IsoStack socket front-end on application cores.

5.2 IsoStack Variants

To analyze the design choices, in particular those related to queuing and aggregation mechanisms, we implemented different variants of the IsoStack:

- Iso-Kernel. This implementation is described in Section 4. In particular, it supports transmit data aggregation, and uses in-kernel per-CPU notification queues; the socket back-end polls only the notification queues.
- Iso-Basic. Each application thread has a separate command/status queue in user-mode. No aggregation is used; each socket command translates to a message in the command queue. The socket back-end polls all the command queues.
- Iso-Aggregated. Uses the same queue structure as the Iso-Basic; implements client and server side transmit data aggregation.
- Iso-Lock. This variant is similar to Iso-Kernel; it reintroduces some of the locks that were eliminated in the other variants. The sole purpose of this variant is to evaluate the impact of un-contended locks, by an experiment described in Section 5.5.

The Iso-Kernel variant is the implementation that we used for most tests. In the rest of this section, unless stated otherwise, the term "IsoStack" refers to Iso-Kernel variant.

5.3 Throughput Evaluation

We used a multi-threaded TTCP-like application to evaluate basic data streaming. We measured the achieved throughput, and the total CPU utilization for all processors.

Since maximal throughput of a single connection is limited by end-to-end TCP behavior, the merit of IsoStack becomes more evident as more TCP connections are used. When the traffic amount is low, the socket back-end dedicated CPU is underutilized, and most of its cycles are wasted on polling empty queues. The observed results in many of the tests with low number of connections showed that the overall machine CPU utilization with the IsoStack implementation is higher compared to the native stack. However, when the number of connections starts to grow, this effect is quickly mitigated and the IsoStack shows not just an increased or identical bandwidth, but also lower CPU utilization.

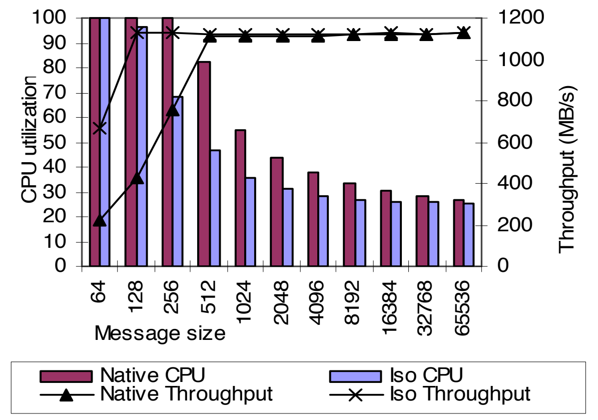


Figure 4. Receive performance for 64 connections

Figure 4 demonstrates receive performance for different message sizes for 64 connections (and 64 application threads). For small messages (64 bytes or 128 bytes), the IsoStack achieves bandwidth that is about 300% better than native, while both systems use almost all available CPU cycles. Clearly, CPU cycles are better used when CPUs are asymmetrically divided between the applications' CPUs and TCP. As message sizes increase, both stacks achieve the line speed with declining CPU utilization, although the native stack still uses more CPU cycles than the IsoStack to drive the same bandwidth. For message sizes above 16 KB, the performance improvement is less prominent: the throughput remains maximal for both stacks, CPU utilization of the IsoStack appears constant (although in fact the dedicated CPU spends more time in polling empty queues), and the CPU utilization of the native stack decreases, as there are fewer system calls for the same amount of data.

Figure 5 demonstrates the transmit performance for different message sizes using 128 connections. The IsoStack reaches the line speed even for a message size as small as 64 bytes, whereas the native stack can reach the line speed only for message sizes of 16 KB and above. Moreover, the IsoStack utilizes far fewer CPU cycles than the native stack. The difference is more dramatic for small messages, where the native stack uses 200% more CPU cycles (while driving a fraction of throughput) than the IsoStack. However, the difference is still high even for large message sizes, when both stacks achieve close to line-speed throughput and the native stack consumes 50% more CPU cycles when compared to the IsoStack.

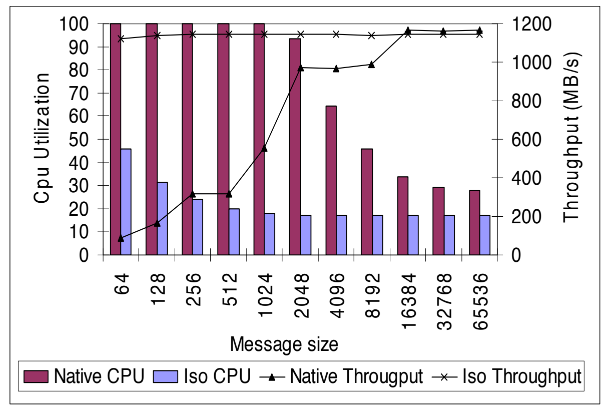


Figure 5. Transmit performance for 128 connections

5.4 Request-Response Performance

In this section, we discuss performance of request/response workloads. Each of the test application threads repeatedly sends and receives a single message, simulating typical client-server communication pattern. This type of workload maximizes the overhead for delegating socket operations to the IsoStack, since each socket operation involves interaction with the stack as no aggregation is taking place.

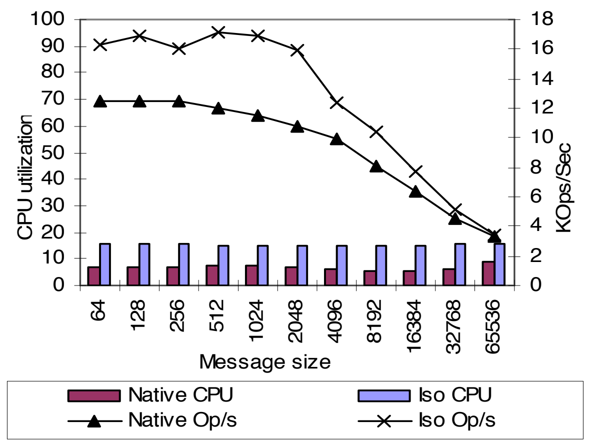


Figure 6. Request-Response test, one connection

Figure 6 demonstrates the request-response test performance for different message sizes using one connection. This allowed us to focus on the impact of socket delegation, without any additional improvements due to aggregation or reduced contention. In this scenario, the IsoStack provides more operations per second for all message sizes, although the difference between the stacks diminishes as the message size increases. Thus, the average latency of a single request-response transaction improves when the IsoStack is used, which may seem surprising because of the added latency imposed by interaction between socket front-end and socket back-end. However, this additional latency of socket delegation is offset by the decreased latency of the network processing, due to lock-free and interrupt-free operation.

Because of the synchronous nature of this test (with just one operation in-flight), the performance is very low for both stacks, due to the delay caused by waiting to the remote application. The CPU utilization for the IsoStack appears to be higher than that of the native stack, since the socket server CPU – although underutilized – still uses 100% of its resources due to wasted polling cycles.

To test the system scalability under the request/response workload, we ran the request-response test with varying numbers of connections (or, equivalently, application threads). Figure 7 shows the CPU utilization and the number of operations per second of both native stack and IsoStack, for different connection numbers, using a message size of 1KB. For up to eight connections, the native stack and IsoStack achieve a similar number of operations per second. For a higher number of connections, the IsoStack CPU becomes fully utilized, and turns into a bottleneck. The native stack allows multiple threads to utilize all processors in the system, and each socket call is executed immediately, even if relatively slowly, on the calling processor. On the other hand, the IsoStack forces serialized execution of socket operations invoked for different sockets on different processors, and thus induces a queuing delay when many processors submit their operations in parallel. Thus, the native stack is able to make progress on each connection faster than the IsoStack, even though its CPU utilization per operation is higher.

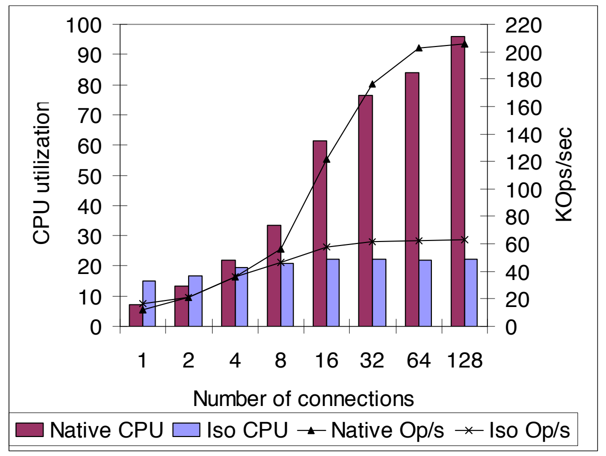


Figure 7. Request-Response test, 1KByte messages

To analyze further the bottleneck imposed by the IsoStack, we measured various code paths inside the socket back-end CPU. We found that simply issuing the kernel call that wakes up the socket application (waiting to receive data) takes approximately 3µs. To compare, the optimized TCP send operation (involving TCP, IP, and MAC layers) also takes approximately 3µs, the socket back-end operation (without the wakeup) takes less than 1µs, and the whole request/response transaction accounts for approximately 16µs. Analysis of the wakeup call shows that the problem is mainly due to contention on several scheduler locks. This indicates that the IsoStack performance could be improved further if a more efficient wakeup mechanism is used.

5.5 Impact of Uncontended Locks

It is a popular belief that reducing lock contention is sufficient to address the problem of the lock overhead. Our implementation went one step further, and eliminated some of the locks completely, avoiding the lock operations altogether for the locks that are only taken on the IsoStack processor. To evaluate the impact of this optimization, we tested an additional variant of the IsoStack, called Iso-Lock, in which we re-instantiated some of the locks – even though they are not needed in our architecture and are only accessed by the IsoStack CPU.

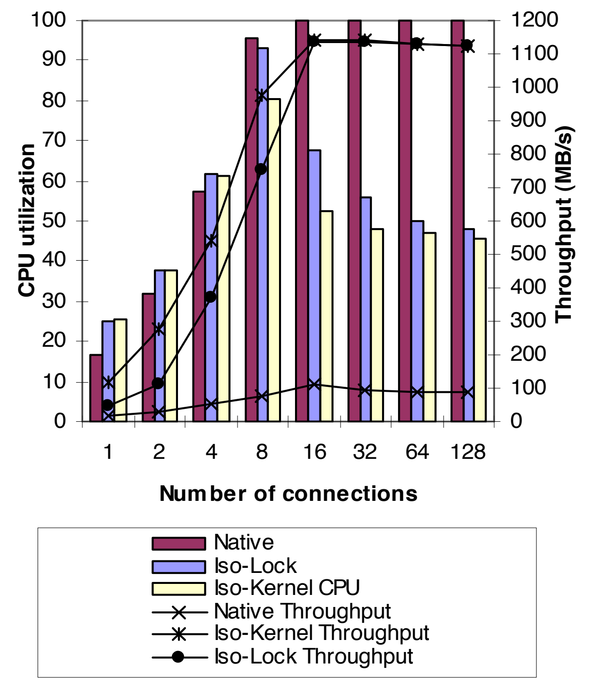


Figure 8. Impact of extra lock on transmit performance for 64 byte messages

Figure 8 depicts the effect that re-instantiating the locks had on the IsoStack performance. For this experiment, we re-introduced the HEA device driver TX and RX locks. These locks were acquired and released each time the device driver transmit or receive handler were called. Socket send throughput tests were performed with a fixed message size of 64 bytes and a variable number of connections. We used the native stack results as the baseline. For a small number of connections, the IsoStack achieves superior throughput compared to the Iso-Lock version, while the CPU utilization appears to be the same. The throughput improvement due to the eliminated lock reaches 200MB/s for eight connections. As the connection number increases, both implementations reach line-speed. The CPU utilization of Iso-Lock is higher than the regular IsoStack variant, which means, oddly, that the socket front-end consumes more CPU. This stems from the fact that additional locks (even though uncontended) make the socket back-end CPU perform slower; the socket transmit buffers then fill up more frequently, causing the socket front-end to wait for free space in the TX buffer. As a result, additional CPU cycles are spent on the extra scheduling that is involved in waking up the socket front-end.

This experiment shows clearly that even un-contended locks are a significant source of overhead. This result may seem counterintuitive, as kernel lock implementation usually takes just a few instructions. Indeed, the locking instruction path length is short, and the atomic update instructions are cache-hits. However, the lock implementation is also required to use a memory barrier – *heavy-weight sync* instruction ([34]), which causes long CPU stall.

Since our implementation did not eliminate all locks that became redundant, the remaining locks pose potential for additional improvement.

5.6 Evaluating Different Queuing Mechanisms

In this section, we try to analyze the performance of queuing mechanisms implemented in the different IsoStack variants.

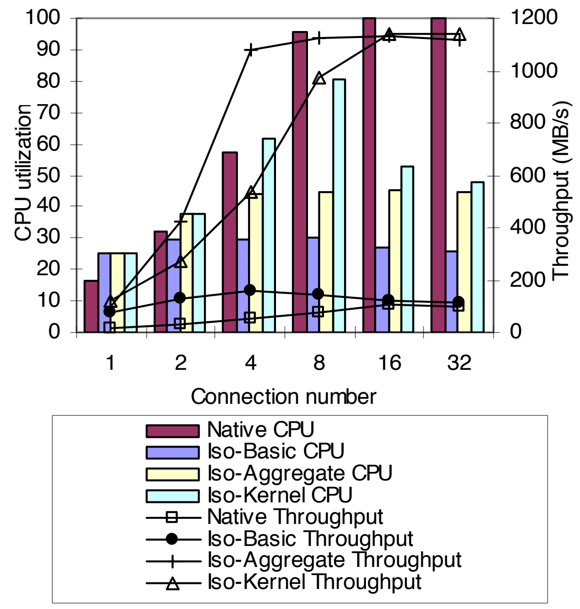


Figure 9. Transmit performance for three IsoStack variants, 64 byte messages

In Figure 9, we compare the 64-byte transmit performance of Iso-Basic (per thread notification queues without aggregation), Iso-Aggregate (per-thread notification queues with aggregation of transmit operations) and Iso-Kernel (per-CPU notification threads with transmit aggregation), with the native stack as a baseline. All three IsoStack variants achieve better throughput with reduced CPU utilization, compared to the native stack. Iso-Aggregate and Iso-Kernel achieve up to eleven times (1000%) more bandwidth than the other variants due to the aggregation that both employ. As a result, they both use more CPU than Iso-Basic, although they still use remarkably less CPU than the native stack. Due to the high cost of using the kernel notification queues, Iso-Aggregate performs better than Iso-Kernel for a low number of connections, but as the number of connections (and application threads) grows, Iso-Aggregate throughput declines, while Iso-Kernel stays at the same throughput with decreased CPU utilization, and eventually out-performs the Iso-Aggregate.

The scalability advantages of the Iso-Kernel variant can be seen more clearly in Figure 10, which depicts the results of a request-response test for varying numbers of connections. The performance of Iso-Aggregate drops dramatically as the number of connections grows beyond 16, while the Iso-Kernel stack scales gracefully, i.e., increased number of clients does not cause performance degradation. This is due to the reduced polling overhead for the socket back-end in the Iso-Kernel implementation, as it polls only the constant number of notification queues, unlike the Iso-Aggregate variant that polls a separate queue for each application thread.

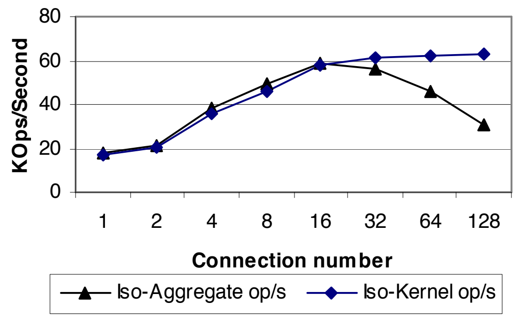


Figure 10. Request/Response Scalabilty

6. Conclusions and Future Work

Our work shows that the design principles of asynchronous interaction, non-shared state, and non-shared processor resources for demanding tasks can be applied to network stack design, yielding significant performance improvements for most workloads. However, some workloads remain challenging. For example, we encountered scenarios where the serialized execution within the IsoStack introduces additional latency when processing particular events. The dispatching of network events handling is rather unsophisticated in our implementation, where the basic policy arbitration is weighted round-robin between the different event queues. Other arbitration policies need to be evaluated, possibly involving a real-time scheduler. Also, it would be beneficial to identify latency-sensitive flows (automatically or with the help of application-provided quality-of-service hints), and prioritize their handling.

We evaluated the system performance for a 10 Gb/s network port, using a single dedicated processor core. As network speed continues to grow, with emerging support for 40 Gb/s and 100 Gb/s, while the processor speed is not expected to increase, it will soon become necessary to employ multiple cores to handle network traffic for a single port in parallel. Fortunately, multi-queue support and minimal packet classification capabilities, available in state-of-the-art adapters, allow parallelization of network processing without re-introducing dependencies between the processors. The IsoStack can be parallelized using independent stack instances for disjoint subsets of network flows, using separate control data structures, and interacting with the client applications through distinct queues.

Our experience shows that dedicating processor cores to specific tasks can improve the overall system performance and scalability. However, the performance gains come at a price: a significant development effort is needed to integrate "isolated" components successfully within a system that was designed under a completely different paradigm. Our implementation had to refrain from using existing system services, as they brought back the very problems we were trying to solve. We believe these services should not be re-invented for every subsystem that can benefit from isolation; instead, the operating system should provide adequate support for isolated execution. Moreover, the underlying hardware should provide better support for inter-processor communication within the system, to supply a better infrastructure for subsystem isolation.

The implementation described in this paper addresses a single OS environment. However, one of the original goals of this work was to devise an architecture for efficient network virtualization. The general architecture described in [26] allows multiple clients to share an isolated I/O subsystem which runs on a different physical machine in a cluster environment or on a different virtual machine within the same physical system. Ironically, interaction between physical machines over a cluster interconnect turned out to be more efficient than interaction between virtual machines within the same POWER system. To realize the performance potential of the IsoStack for virtualized systems, the hypervisor and the underlying hardware have to provide better support for efficient inter-processor communication between processors assigned to different virtual machines.

7. Acknowledgements

We would like to thank Pratap Pattnaik for the idea to evaluate the IsoStack architecture in AIX operating system. Additionally, we thank Joefon Jann, R. S. Burugula, Tom Mathews, Venkat Venkatsubra, G Shantala, Rakesh Sharma and Dave Marquardt for helpful discussions and for making it possible for us to use the AIX development environment. We also thank Herman Dierks for inspiring discussions on performance evaluation, Alan Jiang for his expert input on AIX services, and Shay Goikhman for his participation in the implementation efforts.

8. References

[1] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. IEEE Communications Magazine, 27(6):23–29, June 1989.

[2] J. Kay and J. Pasquale. The importance of non-data touching processing overheads in TCP/IP. In Proceedings of the SIGCOMM Symposium on Communications Architectures and Protocols, pages 259–268. ACM, September 1993.

[3] P. Buonadonna, A. Geweke, D. Culler. An Implementation and Analysis of the Virtual Interface Architecture, In Proceedings of SuperComputing '98.

[4] S. Muir and J. Smith. Functional divisions in the Piglet multiprocessor operating system. In Eighth ACM SIGOPS European Workshop, September 1998.

[5] J. S. Chase, A. J. Gallatin, and K. G. Yocum. End system optimizations for high-speed TCP. IEEE Communications, Special Issue on High-Speed TCP, 39(4):68–74, April 2001.

[6] E. P. Markatos. Speeding-up TCP/IP: faster processors are not enough. In Proceedings of the 21st IEEE International Performance, Computing, and Communications Conference (IPCCC 2002), April 2002, pages 341-345.

[7] M. Rangarajan, A. Bohra, K. Banerjee, E. V. Carrera, R. Bianchini, L. Iftode, W. Zwaenepoel. TCP Servers: Offloading TCP Processing in Internet Servers–Design, Implementation and Performance. Rutgers University Department of CS TR, DCS-TR-481, 2002.

[8] P. Buonadonna and D. Culler. Queue-pair IP: A hybrid architecture for system area networks. In

Proc. 29th Ann. Int'l Symp. on Computer Architecture, pages 247--256, May 2002.

[9] The Infiniband Trade Association. The Infiniband Architecture. http://www.infinibandta.org/specs.

[10] A. Romanow, and S. Bailey. An Overview of RDMA over IP. In p roceedings of the First International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet 2003), Feburary 2003.

[11] J. Mogul. TCP offload is a dumb idea whose time has come. In Workshop on Hot Topics in Operating Systems (HotOS). May 2003.

[12] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When does hardware support help? In 2nd USENIX Symposium on File and Storage Technologies (FAST), March 2003.

[13] P. Shivam, J. S. Chase. Promises and reality: On the elusive benefits of protocol offload. In ACM SigComm Workshop on Network-IO Convergence (NICELI), 2003.

[14] G. Regnier, D. Minturn, G. McAlpine, V. A. Saletore, A. Foong: ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. A Symposium on High Performance Interconnects (HOT Interconnects), 2003.

[15] D. McAuley and R. Neugebauer. A case for Virtual Channel Processors. In Proceedings of the First Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI), 2003.

[16] A. Foong, T. Huff, H. Hum, J. Patwardhan. TCP Performance Re-Visited. In Proc. 2003 IEEE Int'l Symp. Performance Analysis of Systems and Software (IPASS 03), 2003, pp. 70-79.

[17] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. IEEE Computer, 37(11):48--58, November 2004.

[18] ‘‘Scalable Networking: Eliminating the Receive Processing Bottleneck—Introducing RSS,’’ white paper,WinHEC 2004,Microsoft.

[19] D. Freimuth, E. Hu, J. LaVoie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server Network Scalability and TCP Offload. In USENIX Annual Technical Conference, April 2005.

[20] V. Saletore, P. Stillwell Jr, J. Wiegert, P. Cayton, J. Gray, G. Regnier. Efficient Direct User Level Sockets for an Intel® Xeon™ Processor Based TCP On-Load Engine. In Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, Apr. 2005.

[21] L. Grossman. Large Receive Offload Implementation in Neterion 10 GbE Ethernet Driver. Ottawa Linux Symposium, 2005.

[22] L. Shalev, V. Makhervaks, Z. Machulsky, G. Biran, J. Satran, M. Ben-Yehuda, I. Shimony. Loosely Coupled TCP Acceleration Architecture. In Proceedings of 14th IEEE Symposium on High-Performance Interconnects (HOTI'06), Aug. 2006.

[23] P. Willmann, S. Rixner, and A. L. Cox. An evaluation of network stack parallelization strategies in modern operating systems. In Proceedings of Usenix Annual Technical Conference, June 2006.

[24] S. Larsen, P. Sarangam, R. Huggahalli. Architectural Breakdown of End-to-End Latency in a TCP/IP Network. International Symposium on Computer Architecture and High Performance Computing, 2007.

[25] A. Menon , W. Zwaenepoel, Optimizing TCP receive performance, USENIX 2008 ATC, p.85-98, June 2008.

[26] J. Satran, L. Shalev, M. Ben-Yehuda, Z. Machulsky. Scalable I/O - A Well-Architected Way to Do Scalable, Secure and Virtualized I/O. In Proceedings of Workshop on I/O Virtualization, 2008.

[27] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, p. 43–57, Dec. 2008.

[28] J. Liu, B. Abali. Virtualization polling engine (VPE): using dedicated CPU cores to accelerate I/O virtualization. ICS 2009: 225-234

[29] 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 Proc. ACM Symposium on OS Principles, Oct. 2009.

[30] LPAR, http://en.wikipedia.org/wiki/LPAR, retrieved on December 22, 2009.

[31] Large segment offload, http://en.wikipedia.org/wiki/Large_segment_offload, retrieved on December 22, 2009.

[32] Nmon, http://en.wikipedia.org/wiki/Nmon, retrieved on December 22, 2009.

[33] Extended Sockets API, www.opengroup.org, retrieved on December 22, 2009.

[34] Power ISA, http://www.power.org/home.

A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility[*]

Xin Li Michael C. Huang Kai Shen Lingkun Chu
University of Rochester *Ask.com*
{xinli@ece, mihuang@ece, kshen@cs}.rochester.edu lchu@ask.com

Abstract

Memory hardware reliability is an indispensable part of whole-system dependability. This paper presents the collection of realistic memory hardware error traces (including transient and non-transient errors) from production computer systems with more than 800 GB memory for around nine months. Detailed information on the error addresses allows us to identify patterns of single-bit, row, column, and whole-chip memory errors. Based on the collected traces, we explore the implications of different hardware ECC protection schemes so as to identify the most common error causes and approximate error rates exposed to the software level.

Further, we investigate the software system susceptibility to major error causes, with the goal of validating, questioning, and augmenting results of prior studies. In particular, we find that the earlier result that most memory hardware errors do not lead to incorrect software execution may not be valid, due to the unrealistic model of exclusive transient errors. Our study is based on an efficient memory error injection approach that applies hardware watchpoints on hotspot memory regions.

1 Introduction

Memory hardware errors are an important threat to computer system reliability [37] as VLSI technologies continue to scale [6]. Past case studies [27,38] suggested that these errors are significant contributing factors to whole-system failures. Managing memory hardware errors is an important component in developing an overall system dependability strategy. Recent software system studies have attempted to examine the impact of memory hardware errors on computer system reliability [11, 26] and security [14]. Software system countermeasures to these errors have also been investigated [31].

Despite its importance, our collective understanding about the rate, pattern, impact, and scaling trends of memory hardware errors is still somewhat fragmented and incomplete. The lack of knowledge on realistic errors has forced failure analysis researchers to use synthetic error models that have not been validated [11, 14, 24, 26, 31]. Without a good understanding, it is tempting for software developers in the field to attribute (often falsely) non-deterministic system failures or rare performance anomalies [36] to hardware errors. On the other hand, anecdotal evidence suggests that these errors are being encountered in the field. For example, we were able to follow a Rochester student's failure report and identify a memory hardware error on a medical System-on-Chip platform (Microchip PIC18F452). The faulty chip was used to monitor heart rate of neonates and it reported mysterious (and alarming) heart rate drops. Using an in-circuit debugger, we found the failure was caused by a memory bit (in the SRAM's 23rd byte) stuck at '1'.

Past studies on memory hardware errors heavily focused on transient (or soft) errors. While these errors received a thorough treatment in the literature [3,30,41,42, 44], non-transient errors (including permanent and intermittent errors) have seen less attention. The scarcity of non-transient error traces is partly due to the fact that collecting field data requires access to large-scale facilities and these errors do not lend themselves to accelerated tests as transient errors do [44]. The two studies of non-transient errors that we are aware of [10, 35] provide no result on specific error locations and patterns.

In an effort to acquire valuable error statistics in real-world environments, we have monitored memory hardware errors in three groups of computers—specifically, a rack-mounted Internet server farm with more than 200 machines, about 20 university desktops, and 70 PlanetLab machines. We have collected error tracking results on over 800 GB memory for around nine months. Our error traces are available on the web [34]. As far as we know, they are the first (and so far only) publicly available memory hardware error traces with detailed error addresses and patterns.

One important discovery from our error traces is that non-transient errors are at least as significant a source of reliability concern as transient errors. In theory, permanent hardware errors, whose symptoms persist over

[*]This work was supported in part by the National Science Foundation (NSF) grants CNS-0615045, CCF-0937571, CCF-0747324, and CNS-0719790. Kai Shen was also supported by an NSF CAREER Award CCF-0448413 and an IBM Faculty Award.

time, are easier to detect. Consequently they ought to present only a minimum threat to system reliability in an ideally-maintained environment. However, some non-transient errors are intermittent [10] (*i.e.*, whose symptoms are unstable at times) and they are not necessarily easy to detect. Further, the system maintenance is hardly perfect, particularly for hardware errors that do not trigger obvious system failures. Given our discovery of non-transient errors in real-world production systems, a holistic dependability strategy needs to take into account their presence and error characteristics.

We conduct trace-driven studies to understand hardware error manifestations and their impact on the software system. First, we extrapolate the collected traces into general statistical error manifestation patterns. We then perform Monte Carlo simulations to learn the error rate and particularly error causes under different memory protection mechanisms (*e.g.*, single-error-correcting ECC or stronger Chipkill ECC [12]). To achieve high confidence, we also study the sensitivity of our results to key parameters of our simulation model.

Further, we use a virtual machine-based error injection approach to study the error susceptibility of real software systems and applications. In particular, we discovered the previous conclusion that most memory hardware errors do not lead to incorrect software execution [11,26] is inappropriate for non-transient memory errors. We also validated the failure oblivious computing model [33] using our web server workload with injected non-transient errors.

2 Background

2.1 Terminology

In general, a fault is the cause of an error, and errors lead to service failures [23]. Precisely defining these terms ("fault", "error", and "failure"), however, can be "surprisingly difficult" [2], as it depends on the notion of the system and its boundaries. For instance, the consequence of reading from a defective memory cell (obtaining an erroneous result) can be considered as a *failure* of the memory subsystem, an *error* in the broader computer system, or it may not lead to any failure of the computer system at all if it is masked by subsequent processing. In our discussion, we use error to refer to the incidence of having incorrect memory content. The root cause of an error is the fault, which can be a particle impact, or defects in some part of the memory circuit. Note that an error does not manifest (*i.e.*, it is a *latent error*) until the corrupt location is accessed.

An error may involve more than a single bit. Specifically, we count all incorrect bits due to the same root cause as part of one error. This is different from the concept of a multi-bit error in the ECC context, in which case the multiple incorrect bits must fall into a single ECC word. To avoid confusions we call these errors word-wise multi-bit instead.

Transient memory errors are those that do not persist and are correctable by software overwrites or hardware scrubbing. They are usually caused by temporary environmental factors such as particle strikes from radioactive decay and cosmic ray-induced neutrons. *Non-transient* errors, on the other hand, are often caused (at least partially) by inherent manufacturing defect, insufficient burn-in, or device aging [6]. Once they manifest, they tend to cause more predictable errors as the deterioration is often irreversible. However, before transitioning into permanent errors, they may put the device into a marginal state causing apparently *intermittent* errors.

2.2 Memory ECC

Computer memories are often protected by some form of *parity-check code*. In a parity-check code, information symbols within a word are processed to generate *check* symbols. Together, they form the coded word. These codes are generally referred to as ECC (error correcting code). Commonly used ECC codes include SECDED and chipkill.

SECDED stands for *single-error correction, double-error detection*. Single error correction requires the code to have a Hamming distance of at least 3. In binary codes, it can be easily shown that r bits are needed for $2^r - 1$ information bits. For double-error detection, one more check bit is needed to increase the minimum distance to 4. The common practice is to use 8 check bits for 64 information bits forming a 72-bit ECC word as these widths are used in current DRAM standards (*e.g.*, DDR2).

Chipkill ECC is designed to tolerate word-wise multi-bit errors such as those caused when an entire memory device fails [12]. Physical constraints dictate that most memory modules have to use devices each providing 8 or 4 bits to fill the bus. This means that a chip-fail tolerant ECC code needs to correct 4 or 8 adjacent bits. While correcting multi-bit errors in a word is theoretically rather straightforward, in practice, given the DRAM bus standard, it is most convenient to limit the ECC word to 72 bits, and the 8-bit parity is insufficient to correct even a 4-bit symbol. To address this issue, one practice is to reduce the problem to that of single-bit correction by spreading the output of, say, 4 bits to 4 independent ECC words. The trade-off is that a DIMM now only provides 1/4 of the bits needed to fill the standard 64-data-bit DRAM bus, and thus a system needs a minimum of 4 DIMMs to function. Another approach is to use *b-adjacent* codes with much more involved matrices for parity generation and checking [7]. Even in this case, a typical implementation requires a minimum of 2 DIMMs. Due to these practical issues, chipkill ECC remains a technique used primarily in the server domain.

3 Realistic Memory Error Collection

Measurement results on memory hardware errors, particularly transient errors, are available in the literature. Ziegler *et al.* from IBM suggested that cosmic rays may cause transient memory bit flips [41] and did a series of measurements from 1978 to 1994 [30,42,44]. In a 1992 test for a vendor 4Mbit DRAM, they reported the rate of 5950 failures per billion device-hour. In 1996, Normand reported 4 errors out of 4 machines with a total of 8.8 Gbit memory during a 4-month test [29]. Published results on non-transient memory errors are few [10,35] and they provide little detail on error addresses and patterns, which are essential for our analysis.

To enable our analysis on error manifestation and software susceptibility, we make efforts to collect realistic raw error rate and patterns on today's systems. Specifically, we perform long-term monitoring on large, non-biased sets of production computer systems. Due to the rareness of memory hardware errors, the error collection can require enormous efforts. The difficulty of acquiring large scale error data is aggravated by the efforts required for ensuring a robust and consistent collection/storage method on a vast number of machines. A general understanding of memory hardware errors is likely to require the collective and sustained effort from the research community as a whole. We are not attempting such an ambitious goal in this study. Instead, our emphasis is on the *realism* of our production system error collection. As such, we do not claim general applicability of our results.

Many large computer systems support various forms of error logging. Although it is tempting to exploit these error logs (as in some earlier study [35]), we are concerned with the statistical consistency of such data. In particular, the constantly improving efficacy of the error statistics collection can result in higher observed error rates over time by detecting more errors that had been left out before, while there could be no significant change in the real error rates. Also, a maintenance might temporarily suspend the monitoring, which will leave the faulty devices accumulate and later swarm in as a huge batch of bad chips once the monitoring comes back online. These factors all prevent a consistent and accurate error observation.

To ensure the statistical consistency of collected data, we perform proactive error monitoring under controlled, uniform collection methodology. Specifically, we monitor memory errors in three environments—a set of 212 production machines in a server farm at Ask.com [1], about 20 desktop computers at Rochester computer science department, and around 70 wide-area-distributed PlanetLab machines. Preliminary monitoring results (of shorter monitoring duration, focusing exclusively on transient errors, with little result analysis) were reported in another paper [25]. Here we provide an overview of our latest monitoring results on all error types. Due to factors such as machine configuration, our access privileges, and load, we obtained uneven amount of information from the three error monitoring environments. Most of our results were acquired from the large set of server farm machines, where we have access to the memory chipset's internal registers and can monitor the ECC-protected DRAM of all machines continuously. Below we focus our result reporting on the data obtained in this environment.

All 212 machines from the server farm use Intel E7520 chipset as memory controller hub [20]. Most machines have 4 GB DDR2 SDRAM. Intel E7520 memory controller is capable of both SECDED or Chipkill ECC. In addition to error detection and correction, the memory controller attempts to log some information about memory errors encountered. Unfortunately, this logging capability is somewhat limited—there are only two registers to track the addresses of two distinct errors. These registers will only capture the first two memory errors encountered. Any subsequent errors will not be logged until the registers are reset. Therefore, we periodically (once per hour) probe the memory controller to read out the information and reset the registers. This probing is realized through enhancements of the memory controller driver [5], which typically requires the administrative privilege on target machines.

Recall that when a memory cell's content is corrupted (creating a latent error), the error will not manifest to our monitoring system until the location is accessed. To help expose these latent errors, we enable hardware memory scrubbing—a background process that scans all memory addresses to detect and correct errors. The intention is to prevent errors from accumulating into more severe forms (*e.g.*, multi-bit) that are no longer correctable. It is typically performed at a low frequency (*e.g.*, 1.5 hours for every 1 GB) [20] to minimize the energy consumption and contention with running applications. Note that scrubbing does not help expose faults—writing varying values into memory does that. Since we monitored the machines for an extended period of time (9 months), the natural usage of the machines is likely to have exposed most (if not all) faults.

We collected error logs for a period of approximately 9 months (from November 30, 2006 to September 11, 2007). In the first 2 months we observed errors on 11 machines. No new errors were seen for 6 months and then 1 more erroneous machine appeared in the most recent

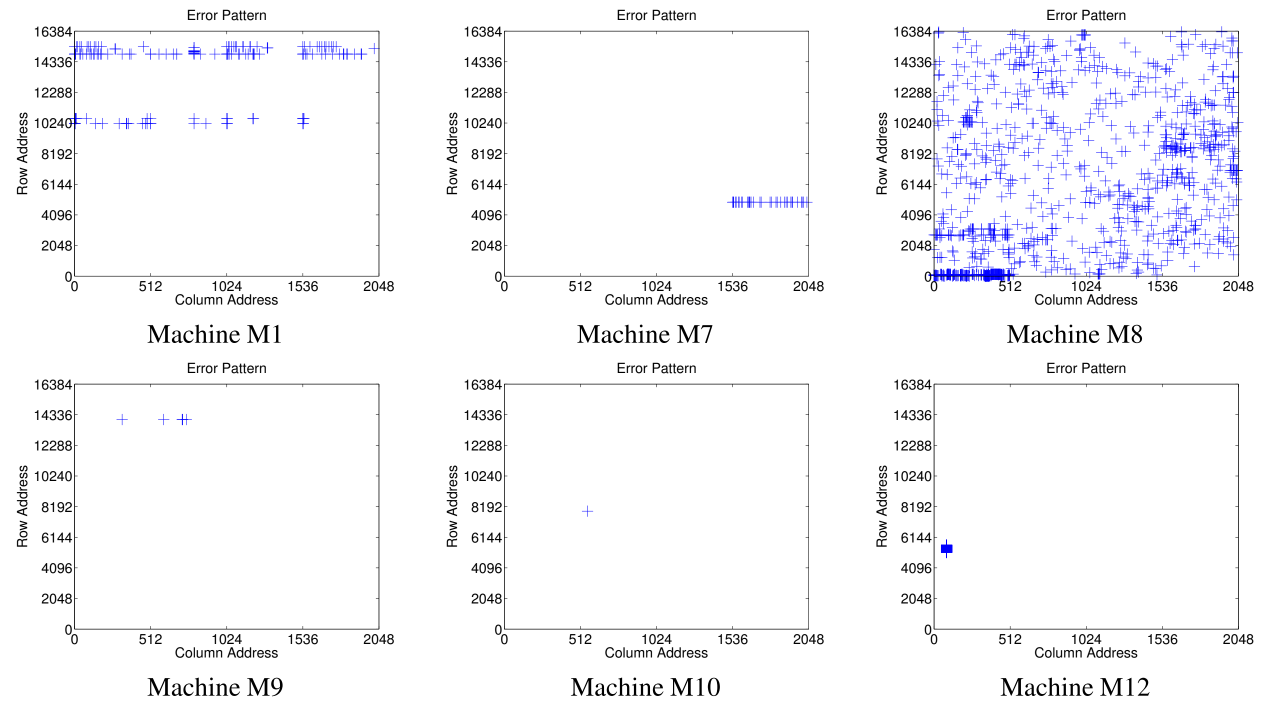


Figure 1. The visualization of example error patterns on physical memory devices. Each cross represents an erroneous cell at its row/column addresses. The system address to row/column address translation is obtained from the official Intel document [20].

month of monitoring. We choose 6 erroneous machines with distinct error patterns and show in Figure 1 how the errors are laid out on the physical memory arrays. Based on the observed patterns, all four memory error modes (single-cell, row, column, and whole-chip [4]) appear in our log. Specifically, M10 contains a single cell error. M7 and M12 represent a row error and a column error respectively. The error case on M1 is comprised of multiple row and columns. Finally, for machine M8, the errors are spread all over the chip which strongly suggests faults in the chip-wide circuitry rather than individual cells, rows, or columns. Based on the pattern of error addresses, we categorize all error instances into appropriate modes shown in Table 1.

While the error-correction logic can detect errors, it cannot tell whether an error is transient or not. We can, however, make the distinction by continued observation—repeated occurrences of error on the same address are virtually impossible to be external noise-induced transient errors as they should affect all elements with largely the same probability. We can also identify non-transient errors by recognizing known error modes related to inherent hardware defects: single-cell, row, column, and whole-chip [4]. For instance, memory row errors will manifest as a series of errors with addresses on the same row. Some addresses on this row may be caught on the log only once. Yet, the cause of that error is most likely non-transient if other cells on the same row indicate non-transient errors (logged multiple times). Consider M9 in Figure 1 as an example, there are five distinct error addresses recorded in our trace, two of which showed up only once while the rest were recorded multiple times. Since they happen on the same row, it is highly probable that they are all due to defects in places like the word line. We count them as a row error.

4 Error Manifestation Analysis

We analyze how device-level errors would be exposed to software. We are interested in the error manifestation rates and patterns (*e.g.*, multi-bits or single-bit) as well as leading causes for manifested errors. We explore results under different memory protection schemes. This is useful since Chipkill ECC represents a somewhat extreme trade-off between reliability and other factors (*e.g.*, performance and energy consumption) and may remain a limited-scope solution. In our memory chipset (Intel E7520) for example, to provide the necessary word length, the Chipkill design requires two memory channels to operate in a lock-stepping fashion, sacrificing throughput and power efficiency.

Machine	Cell	Row	Column	Row-Column	Whole-Chip
M1				1	
M2		1			
M3	1 (transient)				
M4	1				
M5	1 (transient)				
M6					1
M7		1			
M8					1
M9		1			
M10	1				
M11	1				
M12			1		
Total	5 (2 transient)	3	1	1	2

Table 1. Collected errors and their modes (single-cell, row, column, multiple rows and columns, or whole-chip errors). Two of the collected errors are suspected to be transient. Over a nine-month period, errors were observed on 12 machines out of the full set of 212 machines being monitored.

DRAM technology	DDR2
DIMM No. per machine	4
Device No. per DIMM	18
Device data width	x4
Row/Column/Bank No.	$2^{14}/2^{11}/4$
Device capacity	512 Mb
Capacity per machine	4 GB
ECC capability	None, SECDED, or Chipkill

Table 2. Memory configuration for our server farm machines.

4.1 Evaluation Methodology

We use a discrete-event simulator to conduct Monte-Carlo simulations to derive properties of manifested errors. We simulate 500 machines with the exact configuration as the Ask.com servers in Section 3. The detailed configuration is shown in Table 2. We first use the error properties extracted from our data to generate error instances in different memory locations in the simulated machines. Then we simulate different ECC algorithms to obtain a trace of manifested memory errors as the output. Our analysis here does not consider software susceptibility to manifested errors, which will be examined in Section 5. Below, we describe several important aspects of our simulation model, including temporal error distributions, device-level error patterns, and the repair maintenance model.

Temporal error distributions— We consider transient and non-transient errors separately in terms of temporal error distribution. Since transient errors are mostly induced by random external events, it is well established that their occurrences follow a memoryless exponential distribution. The cumulative distribution function of exponential distribution is $F(t) = 1 - e^{-\lambda t}$, which represents the probability that an error has already occurred by time t. The instantaneous error rate for exponential distribution is constant over time, and does not depend on how long the chip has been operating properly.

The non-transient error rate follows a "bathtub" curve with a high, but declining rate in the early "infant mortality" period, followed by a long and stable period with a low rate, before rising again when device wear-out starts to take over. Some study has also suggested that improved manufacturing techniques combined with faster upgrade of hardware have effectively made the wear-out region of the curve irrelevant [28]. In our analysis, we model 16 months of operation and ignore aging or wear-out. Under these assumptions, we use the oft-used Weibull distributions which has the following cumulative distribution function: $F(t) = 1 - e^{(t/\beta)^{\alpha}}$. The *shape parameter* α controls how steep the rate decreases, and the *scale parameter* β determines how "stretched out" the curve is. Without considering the wear-out region, the shape parameter in the Weibull distribution is no more than 1.0, at which point the distribution degenerates into an exponential distribution. The temporal error occurrence information in our data suggested a shape parameter of 0.11.

Device-level error patterns— For transient errors, prior studies and our own observation all point to the single-bit pattern. For non-transient errors, we have the 10 distinct patterns in our trace as templates. When a non-transient error is to be generated, we choose one out of these templates in a uniformly random fashion. There is a problem associated with using the exact template patterns—error instances generated from the same templates are always injected on the same memory location and thus they would always be aligned together to cause an uncorrectable error in the presence of ECC. To address this problem, we shift the error location by a random offset each time we inject an error instance.

Repair maintenance model— Our model requires a faulty device repair maintenance strategy. We employ an idealized "reactive" repair without preventive maintenance. We assume an error is detected as soon as it is exposed to the software level. If the error is diagnosed to be non-transient, the faulty memory module is replaced. Otherwise the machine will undergo a reboot. In our exploration, we have tried two other maintenance models that are more proactive. In the first case, hardware scrubbing is turned on so that transient errors are automatically corrected. In the second case, we further assume that the memory controller notifies the user upon detecting a correctable non-transient error so that faulty memory modules can be replaced as early as possible. We found these preventive measures have a negligible impact on our results. We will not consider these cases in this paper.

Below, Section 4.2 provides evaluation results using the above described model. Due to the small number of errors in the collected error trace, the derived rate and temporal manifestation pattern may not provide high statistical confidence. To achieve high confidence, we further study the sensitivity of our results to two model parameters—the Weibull distribution shape parameter for non-transient errors (Section 4.3) and the temporal error rate (Section 4.4).

4.2 Base Results

Here we present the simulation results on failures. The failure rates are computed as the average of the simulated operational duration. We describe our results under different memory protection schemes.

Figure 2(A) illustrates the failure rates and the breakdown of the causes when there is no ECC protection. In this case, any error will be directly exposed to software and cause a failure. As a result, we can study the errors in isolation. With our measurement, the transient error rate is 2006 FIT [1] for each machine's memory system. Depending on the operational time of the machines, the average non-transient error rates would vary, and so are the corresponding failure rates. Overall, for machines without ECC support, both transient and non-transient errors contribute to the overall error rate considerably.

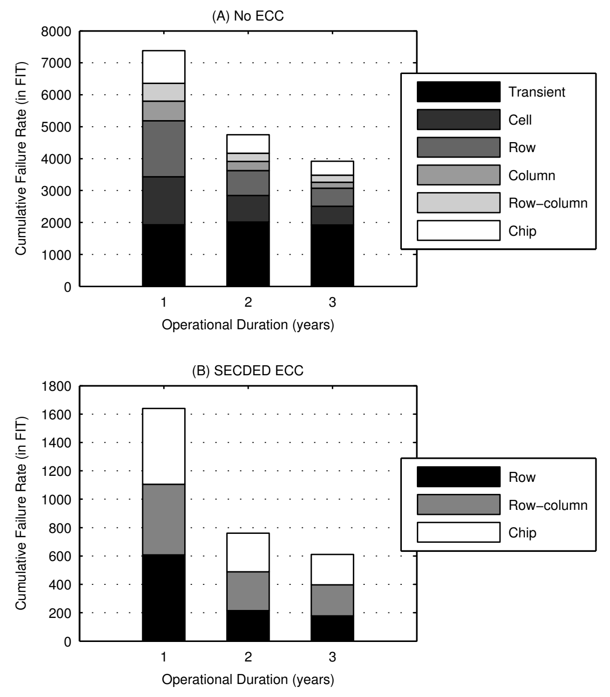


Figure 2. Base failure rates and breakdown causes for no ECC and SECDED ECC. Results (with varying machine operational durations) are for Section 4.2.

SECDED ECC can correct word-wise single-bit errors. For the errors in our trace, it could correct all but one whole-chip error, one row error, and one row-column error. These three cases all have multiple erroneous bits (due to the same root cause) in one ECC word, preventing ECC correction. Theoretically, a failure can also occur when multiple independent single-bit errors happen to affect the same ECC word (such as when a transient error occurs to an ECC word already having a single-bit non-transient error). However, since errors are rare in general, such combination errors are even less probable. In our simulations, no such instance has occurred. Figure 2(B) summarizes the simulation results.

When using the Chipkill ECC, as expected, the memory system becomes very resilient. We did not observe any uncorrected errors. This result echoes the conclusion of some past study [12].

[1] FIT is a commonly used unit to measure failure rates and 1 FIT equals one failure per billion device-hour. To put the numbers into perspectives, IBM's target FIT rates for servers are 114 for undetected (or silent) data corruption, 4500 for detected errors causing system termination, and 11400 for detected errors causing application termination [8]. Note that these rates are for the whole system including all components.

4.3 Shape Parameter Sensitivity for Non-Transient Error Distribution

To reach high confidence in our results, we consider a wide range of the Weibull shape parameters for the non-transient error temporal distribution and study the sensitivity of our results to this parameter. We use a machine operational duration of 16 months, which is the age of the Ask.com servers at the end of our data collection.

Prior failure mode studies in computer systems [16, 40], spacecraft electronics [17], electron tubes [22], and integrated circuits [19] pointed to a range of shape parameter values in 0.28–0.80. Given this and the fact that the Weibull distribution with shape parameter 1.0 degenerates to an exponential distribution, we consider the shape parameter range of 0.1–1.0 in this sensitivity study.

In both ECC mechanisms, the non-transient error rate depends on the Weibull shape parameter. The lower the shape parameter, the faster the error rate drops, and the lower the total error rate for the entire period observed. Note that the transient error rate also fluctuates a little because of the non-deterministic nature of our Monte-Carlo simulation. But the change of transient error rates does not correlate with the shape parameter. For no-ECC, as Figure 3(A) shows, for machines in their first 16 months of operation, the difference caused by the wide ranging shape parameter is rather insignificant.

In the case of SECDED shown in Figure 3(B), the impact of the Weibull shape parameter is a bit more pronounced than in the case of no ECC but is still relatively insignificant. Also, even though error rates are significantly reduced by SECDED, they are still within a factor of about five from those without ECC.

4.4 Statistical Error Rate Bounds

Due to the small number of device-level errors in our trace, the observed error rate may be quite different from the intrinsic error rate of our monitored system. To account for such inaccuracy, we use the concept of *p-value bounds* to provide a range of possible intrinsic error rates with statistical confidence.

For a given probability p, the p-value upper bound (λ_u) is defined as the intrinsic error rate under which $Pr\{X \leq n\} = p$. Here n is the actual number of errors observed in our experiment. X is the random variable for the number of errors occurring in an arbitrary experiment of the same time duration. And likewise, the p-value lower bound (λ_l) is the intrinsic error rate under which $Pr\{X \geq n\} = p$. A very small p indicates that given n observed errors, it is improbable for the actual intrinsic error rate λ to be greater than λ_u or less than λ_l.

Given p, the probability distribution of random variable X is required to calculate the p-value for our data.

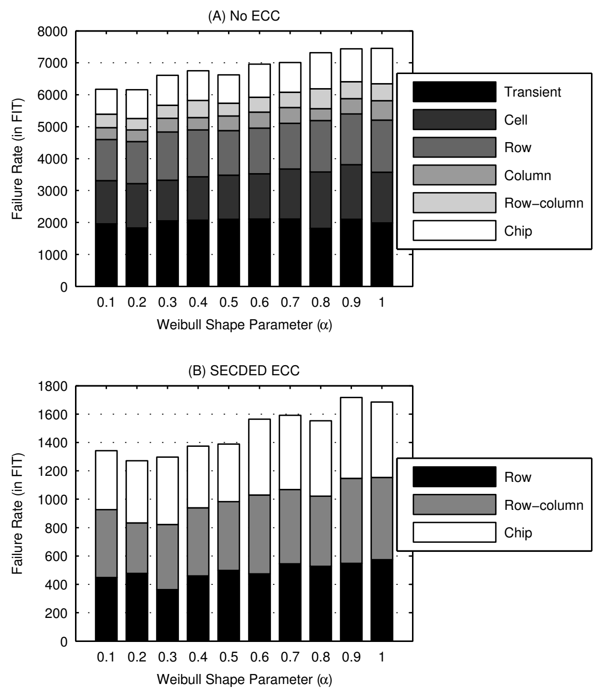


Figure 3. Failure rates and breakdown causes for no ECC and SECDED ECC, with varying Weibull shape parameter for non-transient error distribution. Results are for Section 4.3.

Thankfully, when the memory chips are considered identical, we can avoid this requirement. This is because in any time interval, their probability of having an error is the same, say q. Let N be the total number of memory chips operating, then the actual number of errors happening in this period, X, will be a random variable which conforms to binomial distribution: $P_{N,q}\{X = k\} = \binom{N}{k} q^k (1-q)^{N-k}$. When N is very large (we simulated thousands of chips), we can approximate by assuming N approaches infinity. In this case the binomial distribution will turn into Poisson distribution. For the ease of calculation, we shall use the form of Poisson distribution: $P_\lambda\{X = k\} = \frac{e^{-\lambda}\lambda^k}{k!}$, where $\lambda = q \cdot N$ is the expectation of X.

Based on the analysis above and the observed error rates, we have calculated the 1% upper and lower bounds. For instance, the transient error rate in non-ECC memory system is 2006 FIT as mentioned earlier. The corresponding 1%-upper-bound and 1%-lower-bound are 8429 FIT and 149 FIT respectively. The bounds on the various manifested error rates, derived from different raw error rates, are shown in Figure 4. From left to right, the bars show the 1%-lower-bound, the originally observed rate, and the 1%-upper-bound. As can be seen, for manifestations caused by non-transient

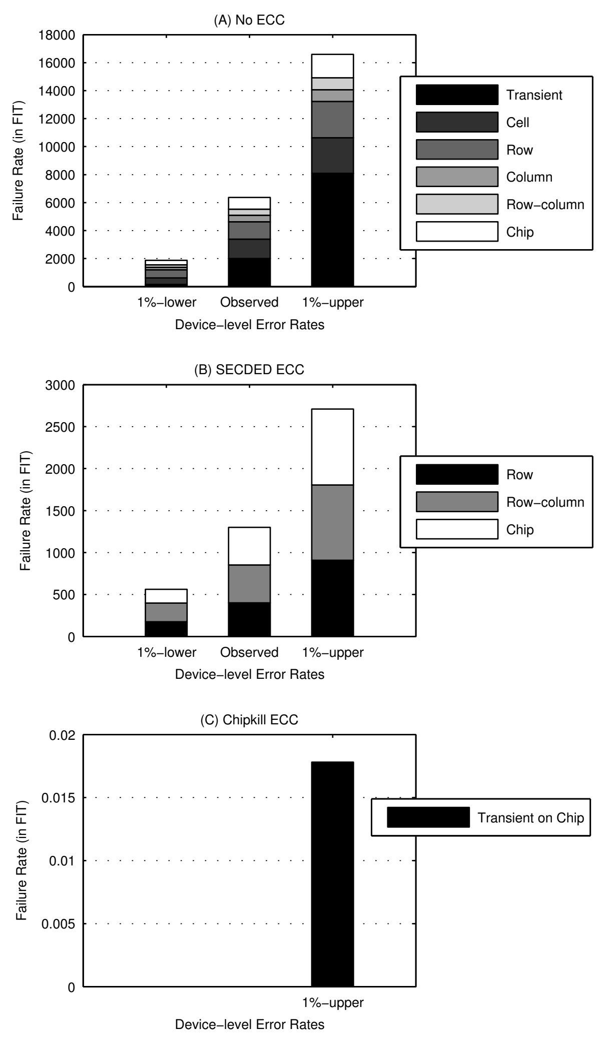


Figure 4. Manifested errors when input device-level error rates are the originally observed and 1%-lower/upper-bounds. Results are for Section 4.4.

errors, the two 1% bounds are roughly 2x to either direction of the observed rate. The ranges are narrow enough such that they have little impact to the qualitative conclusions.

For Chipkill ECC, the 1%-upper-bound offers a better chance to observe failures in the outcome of our simulation. With this increased rate, we finally produced a few failure instances (note there were none for Chipkill in the base simulations done in previous sub-sections). The patterns of the failures are shown in Figure 4(C). All of the failures here are caused by a transient error hitting an existing non-transient chip error.

4.5 Summary

We summarize our results of this part of the study:

- In terms of the absolute failure rate, with no ECC protection, error rates are at the level of thousands of FIT per machine. SECDED ECC lowers the rates to the neighborhood of 1000 FIT per machine. Chipkill ECC renders failure rates virtually negligible.

- Non-transient errors are significant (if not dominant) causes for all cases that we evaluated. Particularly on SECDED ECC machines, manifested failures tend to be caused by row errors, row-column errors, and whole-chip errors. With Chipkill ECC, the few failures occur when a transient error hits a memory device already inflicted with whole-chip non-transient errors.

- In terms of the error patterns, word-wise multi-bit failures are quite common.

Major implications of our results are that memory hardware errors exert non-negligible effects on the system dependability, even on machines equipped with SECDED ECC. Further, system dependability studies cannot assume a transient-error-only or single-bit-error-only model for memory hardware errors.

5 Software System Susceptibility

A memory error that escaped hardware ECC correction is exposed to the software level. However, its corrupted memory value may or may not be consumed by software programs. Even if it is consumed, the software system and applications may continue to behave correctly if such correctness does not depend on the consumed value. Now we shift our attention to the susceptibility of software systems and applications to memory errors. Specifically, we inject the realistic error patterns from our collected traces and observe the software behaviors. Guided by the conclusion of Section 4, we also take into account the shielding effect of ECC algorithms.

There is a rich body of prior research on software system reliability or security regarding memory hardware errors [11, 14, 24, 26, 31, 33]. One key difference between these studies and ours is that all of our analysis and discussions build on the realism of our collected error trace. In this section, we tailor our software susceptibility evaluation in the context of recent relevant research with the hope of validating, questioning, or augmenting prior results.

5.1 Methodology of Empirical Evaluation

Memory Access Tracking and Manipulation To run real software systems on injected error patterns, we must accomplish the following goals. First, every read access to a faulty location must be supplied with an erroneous value following the injection pattern. This can be achieved by writing the erroneous value to each individual faulty address at the time of injection. Second, for every write access to a faulty location, if the error is non-transient, we must guarantee the erroneous value is restored right after the write. The injection is then followed by error manifestation bookkeeping. The bookkeeping facility has to be informed whenever a faulty address is accessed so that it would log some necessary information. The key challenge of such error injection and information logging is to effectively track and manipulate all the accesses to locations injected with errors (or *tracked locations*).

Traditional memory tracking approaches include:

- *Hardware watchpoints* [26]—employing hardware memory watchpoints on tracked locations. Due to the scarcity of hardware watchpoints on modern processors, this approach is not scalable (typically only able to track a few memory locations at a time).

- *Code instrumentation* [39]—modifying the binary code of target programs to intercept and check memory access instructions. This approach may incur excessive overhead since it normally must intercept all memory access instructions before knowing whether they hit on tracked memory locations. Further, it is challenging to apply this approach on whole-system monitoring including the operating system, libraries, and all software applications.

- *Page access control* [39]—applying virtual memory mechanism to trap accesses to all pages containing tracked memory locations and then manipulating them appropriately with accesses enabled. For this approach, it is important to reinstate the page access control after each page fault handling. This is typically achieved by single-stepping each trapped memory access, or by emulating the access within the page fault handler. This approach may also incur substantial overhead on *false memory traps* since all accesses to a page trigger traps even if a single location in the page needs to be tracked.

We propose a new approach to efficiently track a large number of memory locations. Our rationale is that although the whole system may contain many tracked locations exceeding the capacity of available hardware watchpoints, tracked locations within an individual page are typically few enough to fit. Further, the locality of executions suggests a high likelihood of many consecutive accesses to each page. By applying hardware watchpoints on tracked locations within the currently accessed *hot* page, we do not have to incur false traps on accesses to non-tracked locations within this page. At the same time, we enforce access control to other pages containing tracked locations. When an access to one such page is detected, we set the new page as hotspot and switch hardware watchpoint setup to tracked locations within the new page. We call our approach *hotspot watchpoints*. Its efficiency can be close to that of hardware watchpoints, without being subject to its scalability constraint. Note that it is possible that tracked locations within a page still exceed the hardware watchpoint capacity. If such a page is accessed, we fall back to the memory access single-stepping as in the page access control approach.

There is a chance for a single instruction to access multiple pages with tracked locations. For example, an instruction's code page and its data page may both contain tracked locations. If we only allow accesses to one tracked page at a time, then the instruction may trap on the multiple tracked pages alternately without making progress—or a livelock. We detect such livelock by keeping track of the last faulted program counter. Upon entering the page-fault handler, we suspect a livelock if the current faulting program counter address is the same as the last one. In such a situation we fall back to the memory access single-stepping while allowing accesses to multiple tracked pages that are necessary. It is possible that a recurring faulted program counter does not correspond to an actual livelock. In this case, our current approach would enforce an unnecessary instruction single-stepping. We believe such cases are rare, but if needed, we may avoid this slowdown by more precise tracking of execution progresses (*e.g.*, using hardware counters).

We should also mention a relevant memory monitoring technique called SafeMem [32], originally proposed for detecting memory leaks and corruptions. With modest modifications, it may be used for continual memory access tracking as well. SafeMem exploits the memory ECC mechanisms to trap accesses to all cachelines containing tracked locations. Because typical cacheline sizes (32–256 bytes) are smaller than the typical page size of 4 KB, false memory traps (those trapped memory accesses that do not actually hit tracked locations) under cacheline access control can be significantly fewer than that under page access control. Nevertheless, our hotspot watchpoints technique can further reduce the remaining false memory traps. In this case, hardware watchpoints will be set upon tracked locations within the current hot cacheline (or cachelines) instead of the hot page.

Error Monitoring Architecture If the error injection and monitoring mechanisms are built into the target system itself (as in [26]), these mechanisms may not behave reliably in the presence of injected memory errors. To avoid this potential problem, we utilize a virtual machine-based architecture in which the target system runs within a hosted virtual machine while the error injection and monitoring mechanisms are built in the underlying virtual machine monitor. We enable the shadow page table mode in the virtual machine memory management. Error injections only affect the shadow page tables while page tables within the target virtual machine are not affected. We also intercept further page table updates—we make sure whenever our faulty pages are mapped to any process, we will mark the protection bit in the corresponding page table.

In order to understand software system susceptibility to memory hardware errors, we log certain information every time an error is activated. Specifically, we record the access type (read or write), access mode (kernel or user), and the program counter value. For kernel mode accesses, we are able to locate specific operating system functions from the program counter values.

System Setup and Overhead Assessment Our experimental environment employs Xen 3.0.2 and runs the target system in a virtual machine with Linux 2.6.16 operating system. We examine three applications in our test: 1) the Apache web server running the static request portion of the SPECweb99 benchmark with around 2 GB web documents; 2) MCF from SPEC CPU2000—a memory-intensive vehicle scheduling program for mass transportation; and 3) compilation and linking of the Linux 2.6.23 kernel. The first is a typical server workload while the other two are representative workstation workloads (in which MCF is CPU-intensive while kernel build involves significant I/O).

We assess the overhead of our memory access tracking mechanism. We select error pattern M8 (illustrated in Figure 1), the one with most number of faulty bits in our overhead assessment. This error pattern consists of 1053 faulty 32-bit long words scattered in 779 pages, among which 668 pages contain only one erroneous word. Note that in this overhead evaluation, we only specify the spots to be tracked without actually flipping the memory bits. So the correctness of system and application executions should not be affected.

We compare the overhead of our approach to that of page access control. Results in Figure 5 suggest that our approach can significantly reduce the overhead compared to the alternative page access control approach. In particular, for Linux kernel build, our approach can reduce the execution time by almost a factor of four. The efficiency of our hotspot watchpoints approach also makes it a promising technique to support other utilization of memory access tracking [39] beyond the hardware error injection in this paper. Across the three applications, kernel build incurs the greatest amount of slowdown due to memory access tracking. We are able to attribute much (about two thirds) of the slowdown to the kernel function named `vma_merge` whose code section

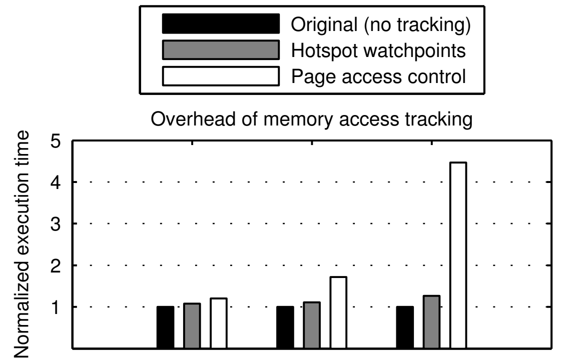


Figure 5. Benchmark execution time of our hotspot watchpoints approach, compared to the page access control approach [39]. The execution time is normalized to that of the original execution without memory access tracking. The slowdown was evaluated with the whole-chip error pattern of M8. Note for web server, the execution time is for requesting all the 2 GB data in a sweep-through fashion.

contains a tracked location. This function is triggered frequently by the GNU compiler when performing memory mapped I/O.

5.2 Evaluation and Discussion

Evaluation on Failure Severity Two previous studies [11, 26] investigated the susceptibility of software systems to transient memory errors. They reached similar conclusions that memory errors do not pose a significant threat to software systems. In particular, Messer *et al.* [26] discovered that of all the errors they injected, on average 20% were accessed, among which 74% are overwritten before being really consumed by the software. In other words, only 5% of the errors would cause abnormal software behaviors. However, these studies limited their scope for single-bit transient errors only. Our findings in Section 4 show non-transient errors are also a significant cause of memory failures. When these errors are taken into account, the previous conclusions may not stand intact. For example, non-transient errors may not be overwritten, and as a result, a portion of the 74% overwritten errors in [26] would have been consumed by the software system if they had been non-transient.

Table 3 summarizes the execution results of our three benchmark applications when non-transient errors are injected. Since our applications all finish in a short time (a few minutes), we consider these non-transient errors as permanent during the execution. In total we had 12 different error patterns. M3 and M5 are transient errors and therefore we do not include them in this result. M8 is so massive that as soon as it is injected, the OS crashes right away. We also exclude it from our results.

Application	Web server	MCF	Kernel build
No ECC			
M1 (row-col error)	WO	AC	AC
M2 (row error)	OK		
M4 (bit error)	OK		
M6 (chip error)	KC	WO	AC
M7 (row error)	WO	WO	
M9 (row error)	OK		
M10 (bit error)	OK		
M11 (bit error)			
M12 (col error)	WO		
SECDED ECC			
M1 (row-col error)	WO	WO	AC
M7 (row error)	WO	WO	

Table 3. Error manifestation for each of our three applications. The abbreviations in the table should be interpreted as follows, with descending manifestation severity: KC—kernel crash; AC—application crash; WO—wrong output; OK—program runs correctly. The blank cells indicate the error was not accessed at all.

Application	Web server	MCF	Kernel build
No ECC			
M1 (row-col error)	WO	AC	OK
M2 (row error)	OK		
M4 (bit error)	OK		
M6 (chip error)	KC	OK	OK
M7 (row error)	WO	OK	
M9 (row error)	OK		
M10 (bit error)	OK		
M11 (bit error)			
M12 (col error)	WO		
SECDED ECC			
M1 (row-col error)	WO	OK	OK
M7 (row error)	WO	OK	

Table 4. Error manifestation for each of our three applications, when the errors are made transient (thus correctable by overwrites). Compared to Table 3, many of the runs are less sensitive to transient errors and exhibit no mis-behavior at the application level.

The table includes results for both cases of no ECC and SECDED ECC. Since errors are extremely rare on Chipkill machines (see conclusions of Section 4), here we do not provide results for Chipkill. For no ECC, briefly speaking, out of the 27 runs, 13 have accessed memory errors and 8 did not finish with expected correct results. This translates to 48% of the errors are activated and 62% of the activated errors do lead to incorrect execution of software systems. In the SECDED case, single-bit errors would be corrected. Most errors (except M1 and M7) are completely shielded by the SECDED ECC. However, for the six runs with error patterns M1 and M7, five accessed the errors and subsequently caused abnormal behaviors.

Overall, compared to results in [26], non-transient errors evidently do cause more severe consequences to software executions. The reason for the difference is twofold— 1) non-transient errors are not correctable by overwriting and 2) unlike transient errors, non-transients sometimes involve a large number of erroneous bits. To demonstrate reason #1, we show in Table 4, when these errors are turned into transient ones (meaning they can be corrected by overwritten values), quite a few of the execution runs would finish unaffected.

Validation of Failure-Oblivious Computing This evaluation study attempts to validate the concept of failure-oblivious computing [33] with respect to memory hardware errors. The failure-oblivious model is based on the premise that in server workloads, error propagation distance is usually very small. When memory errors occur (mostly they were referring to out-of-bound memory accesses), a failure-oblivious computing model would discard the writes and supply the read with arbitrary values and try to proceed. In this way the error occurred will be confined within the local scope of a request and the server computation can be resumed without being greatly affected.

The failure-oblivious concept may also apply to memory hardware errors. It is important to know what the current operating system does in response to memory errors. Without ECC, the system is obviously unaware of any memory errors going on. Therefore it is truly failure-oblivious. With ECC, the system could detect some of the uncorrectable errors. At this point the system can choose to stop, or to continue execution (probably with some form of error logging). The specific choices are configurable and therefore machine dependent.

For our web server workload, we check the integrity of web request returns in the presence of memory errors. Table 5 lists the number of requests with wrong contents for the error cases. We only show error cases that trigger wrong output for the web server (as shown in Table 3). The worst case is M1, which caused 15 erroneous request returns (or files with incorrect content). However, this is still a small portion (about 0.1%) in the total 14400 files we have requested. Our result suggests that, in our tested web server workload, memory-hardware-error-induced failures tend not to propagate very far. This shows the promise of applying failure-oblivious computing in the management of memory hardware errors for server systems.

Discussion on Additional Cases Though error testing data from the industry are seldom published, modern

No ECC

M1 (row-col error)	Wrong output (15 requests)
M7 (row error)	Wrong output (2 requests)
M12 (col error)	Wrong output (1 request)

SECDED ECC

M1 (row-col error)	Wrong output (8 requests)
M7 (row error)	Wrong output (1 request)

Table 5. Number of requests affected by the errors in SPECweb99-driven Apache web server. We only show error cases that trigger wrong output for the web server (as shown in Table 3). We request 14400 files in the experiment.

commercial operating systems do advocate their countermeasures for faulty memory. Both IBM AIX [18] and Sun Solaris [38] have the ability to retire faulty memory when the ECC reports excessive correctable memory errors. Our results suggest that with ECC protection, the chances of errors aligning together to form an uncorrectable one is really low. However, this countermeasure could be effective against those errors that gradually develop into uncorrectable ones by themselves. Since our data does not have timestamps for most of the error instances, it is hard to verify how frequently these errors occur. On Chipkill machines [18], however, this countermeasure seems to be unnecessary since our data shows that without any replacement policy, Chipkill will maintain the memory failure rate at an extremely low level.

A previous security study [14] devised a clever attack that exploits memory errors to compromise the Java virtual machine (JVM). They fill the memory with pointers to an object of a particular class, and through an accidental bit flip, they hope one of the pointers can point to an object of another class. Obtaining a class A pointer actually pointing to a class B object is enough to compromise the whole JVM. In particular, they also provided an analysis of the effectiveness of exploiting multi-bit errors [14]. It appears that they can only exploit bit flips in a region within a pointer word (in their case, bit 2:27 for a 32-bit pointer). In order for an error to be exploitable, all the bits involved must be in the region. The probability that they can exploit the error decreases with the number of erroneous bits in the word. Considering that the multi-bit errors in our collected error trace are mostly consecutive rather than distributed randomly, we can be quite optimistic about successful attacks.

Another previous study [31] proposed a method to protect critical data against illegal memory writes as well as memory hardware errors. The basic idea is that software systems can create multiple copies of their critical data. If a memory error corrupts one copy, a consistency check can detect and even correct such errors. The efficacy of such an approach requires that only one copy of the critical data may be corrupted at a time. Using our collected realistic memory error patterns, we can explore how the placement of multiple critical data copies affects the chance for simultaneous corruption. In particular, about half of our non-transient errors exhibit regular column or row-wise array patterns. Therefore, when choosing locations for multiple critical data copies, it is best to have them reside in places with different hardware row and column addresses (especially row addresses).

6 Related Work

The literature on memory hardware errors can be traced back over several decades. In 1980, Elkind and Siewiorek reported various failure modes caused by low-level hardware fault mechanisms [13]. Due to the rareness of these errors, collecting error samples at a reasonable size would require a substantial amount of time and resource in field tests. Such field measurements have been conducted in the past (most notably by Ziegler at IBM) [29, 30, 43, 45]. These studies, however, have exclusively dedicated to transient errors and single-bit error patterns. Our previous study on transient error rates [25] also falls into this category.

Studies that cover non-transient errors are relatively few. In 2002, Constantinescu [10] reported error collection results on 193 machines. More recently, Schroeder *et al.* [35] examined memory errors on a larger number of servers from six different platforms. The large dataset enabled them to analyze statistical error correlations with environmental factors such as machine temperature and resource utilization. However, these studies provide no detail on error addresses or any criteria for categorizing transient and non-transient errors. Such results are essential for the error manifestation analysis and software susceptibility study in this paper.

Previous research has investigated error injection approaches at different levels. Kanawati *et al.* [21] altered target process images from a separate injection process that controls the target using `ptrace` calls. This is a user-level method that cannot inject errors to the operating system image. Li *et al.* [24] injected errors into hardware units using a whole-system simulator. This approach allows failure analysis over the whole system but the slow simulator speed severely limits the analysis scale.

Several studies utilized debugging registers for error injection at close-to-native speed. Gu *et al.* [15] focused on injecting faults in instruction streams (rather than memory error injection in our study). Carreira *et al.* [9] resorted to external ECC-like hardware to track the activation of memory errors whereas our approach is a software-only approach and therefore it can be applied on off-the-shelf hardware. In addition, they cannot monitor non-transient errors without completely single-stepping the execution. Messer *et al.* [26] also targeted transient errors. And their direct use of the watchpoint registers limited the number of simultaneously injected errors. In contrast, our hotspot watchpoint technique allows us to inject any number of transient and non-transient errors at high speed.

7 Conclusion

Memory hardware reliability is an indispensable part of whole-system dependability. Its importance is evidenced by a plethora of prior studies of memory error's impact on software systems. However, the absence of solid understanding of the error characteristics prevents software system researchers from making well reasoned assumptions, and it also hinders the careful evaluations over different choices of fault tolerance design.

In this paper, we have presented a set of memory hardware error data collected from production computer systems with more than 800 GB memory for around 9 months. We discover a significant number of non-transient errors (typically in the patterns of row or column errors). Driven by the collected error patterns and taking into account various ECC protection schemes, we conducted a Monte Carlo simulation to analyze how errors manifest at the interface between the memory subsystem and software applications. Our basic conclusion is that non-transient errors comprise a significant portion of the overall errors visible to software systems. In particular, with the conventional ECC protection scheme of SECDED, transient errors will be almost eliminated while only non-transient memory errors may affect software systems and applications.

We also investigated the susceptibility of software system and applications to realistic memory hardware error patterns. In particular, we find that the earlier results that most memory hardware errors do not lead to incorrect software execution [11, 26] may not be valid, due to the unrealistic model of exclusive transient errors. At the same time, we provide a validation for the failure-oblivious computing model [33] on a web server workload with injected memory hardware errors. Finally, as part of our software system susceptibility study, we proposed a novel memory access tracking technique that combines hardware watchpoints with coarse-grained memory protection to simultaneously monitor large number of memory locations with high efficiency.

Acknowledgments

We would like to thank Howard David at Intel for kindly interpreting the memory error syndromes discovered in our measurement. We also thank the anonymous USENIX reviewers and our shepherd Shan Lu for helpful comments on a preliminary version of this paper.

References

[1] Ask.com (formerly Ask Jeeves Search). http://www.ask.com.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.

[3] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, May 2005.

[4] M. Blaum, R. Goodman, and R. Mceliece. The reliability of single-error protected computer memories. *IEEE Trans. on Computers*, 37(1):114–118, 1988.

[5] EDAC project. http://bluesmoke.sourceforge.net.

[6] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov.–Dec. 2005.

[7] D. Bossen. *b*-adjacent error correction. *IBM Journal of Research and Development*, 14(4):402–408, 1970.

[8] D. Bossen. CMOS soft errors and server design. In *2002 Reliability Physics Tutorial Notes – Reliability Fundamentals*, pages 121.07.1–121.07.6, Dallas, Texas, Apr. 2002.

[9] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. Software Eng.*, 24(2):125–136, 1998.

[10] C. Constantinescu. Impact of deep submicron technology on dependability of VLSI circuits. In *Int'l Conf. on Dependable Systems and Networks*, pages 205–209, Bethesda, MD, June 2002.

[11] C. da Lu and D. A. Reed. Assessing fault sensitivity in MPI applications. In *Supercomputing*, Pittsburgh, PA, Nov. 2004.

[12] T. J. Dell. A white paper on the benefits of chipkill correct ECC for PC server main memory. *White paper*, 1997.

[13] S. A. Elkind and D. P. Siewiorek. Reliability and performance of error-correcting memory and register arrays. *IEEE Trans. on Computers*, (10):920–927, 1980.

[14] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symp. on Security and Privacy*, pages 154–165, Berkeley, CA, May 2003.

[15] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *Int'l Conf. on Dependable Systems and Networks*, pages 459–468, 2003.

[16] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. In *ACM SIGMETRICS*, pages 217–227, Marina del Rey, CA, June 2002.

[17] H. Hecht and E. Fiorentino. Reliability assessment of spacecraft electronics. In *Annu. Reliability and Maintainability Symp.*, pages 341–346. IEEE, 1987.

[18] D. Henderson, B. Warner, and J. Mitchell. IBM POWER6 processor-based systems: Designed for availability. *White paper*, 2007.

[19] D. P. Holcomb and J. C. North. An infant mortality and long-term failure rate model for electronic equipment. *AT&T Technical Journal*, 64(1):15–31, January 1985.

[20] Intel E7520 chipset datasheet: Memory controller hub (MCH). http://www.intel.com/design/chipsets/E7520_E7320/documentation.htm.

[21] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Trans. Computers*, 44(2):248–260, 1995.

[22] J. H. K. Kao. A graphical estimation of mixed Weibull parameters in life-testing of electron tubes. *Technometrics*, 1(4):389–407, Nov. 1959.

[23] J. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *15th Int'l Symp. on Fault-Tolerant Computing*, pages 2–11, Ann Arbor, MI, June 1985.

[24] M. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–276, Seattle, WA, Mar. 2008.

[25] X. Li, K. Shen, M. Huang, and L. Chu. A memory soft error measurement on production systems. In *USENIX Annual Technical Conf.*, pages 275–280, Santa Clara, CA, June 2007.

[26] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. S. Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Trans. on Computers*, 53(12):1557–1568, 2004.

[27] B. Murphy. Automating software failure reporting. *ACM Queue*, 2(8):42–48, Nov. 2004.

[28] F. R. Nash. *Estimating Device Reliability: Assessment of Credibility*. Springer, 1993. ISBN 079239304X.

[29] E. Normand. Single event upset at ground level. *IEEE Trans. on Nuclear Science*, 43(6):2742–2750, 1996.

[30] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. of Research and Development*, 40(1):41–50, 1996.

[31] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Third EuroSys Conf.*, pages 219–232, Glasgow, Scotland, Apr. 2008.

[32] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *11th Int'l Symp. on High-Performance Computer Architecture*, pages 291–302, San Francisco, CA, Feb. 2005.

[33] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *6th USENIX Symp. on Operating Systems Design and Implementation*, pages 303–316, San Francisco, CA, Dec. 2004.

[34] Rochester memory hardware error research project. http://www.cs.rochester.edu/research/os/memerror/.

[35] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: a large-scale field study. In *ACM SIGMETRICS*, pages 193–204, Seattle, WA, June 2009.

[36] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, pages 85–96, Seattle, WA, June 2009.

[37] Sun Microsystems server memory failures. http://www.forbes.com/global/2000/1113/0323026a.html.

[38] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro. Assessment of the effect of memory page retirement on system RAS against hardware faults. In *Int'l Conf. on Dependable Systems and Networks*, pages 365–370, Philadelphia, PA, June 2006.

[39] R. Wahbe. Efficient data breakpoints. In *5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, Boston, MA, Oct. 1992.

[40] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT system field failure data analysis. In *Pacific Rim Intl. Symp. on Dependable Computing*, pages 178–185, Hong Kong, China, Dec. 1999.

[41] J. Ziegler and W. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(16):776–788, Nov. 1979.

[42] J. Ziegler, M. Nelson, J. Shell, R. Peterson, C. Gelderloos, H. Muhlfeld, and C. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *IEEE Journal of Solid-State Circuits*, 33(2):246–252, Feb. 1998.

[43] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. of Research and Development*, 40(1):19–39, 1996.

[44] J. F. Ziegler et al. IBM experiments in soft fails in computer electronics (1978–1994). *IBM J. of Research and Development*, 40(1):3–18, 1996.

[45] J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, T. J. O'Gorman, and J. M. Ross. Accelerated testing for cosmic soft-error rate. *IBM J. of Research and Development*, 40(1):51–72, 1996.

The Utility Coprocessor: Massively Parallel Computation from the Coffee Shop

John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch
Microsoft Research

Abstract

UCop, the "utility coprocessor," is middleware that makes it cheap and easy to achieve dramatic speedups of parallelizable, CPU-bound desktop applications using utility computing clusters in the cloud. To make UCop performant, we introduced techniques to overcome the low available bandwidth and high latency typical of the networks that separate users' desktops from a utility computing service. To make UCop economical and easy to use, we devised a scheme that hides the heterogeneity of client configurations, allowing a single cluster to serve virtually everyone: in our Linux-based prototype, the only requirement is that users and the cluster are using the same major kernel version.

This paper presents the design, implementation, and evaluation of UCop, employing 32–64 nodes in Amazon EC2, a popular utility computing service. It achieves 6–11× speedups on CPU-bound desktop applications ranging from video editing and photorealistic rendering to strategy games, with only minor modifications to the original applications. These speedups improve performance from the coffee-break timescale of minutes to the 15–20 second timescale of interactive performance.

1 Introduction

The hallmark that separates desktop computing from batch computing is the notion of interactivity: users can see their work in finished form as they go. However, many CPU-intensive applications that are best used interactively, such as video editing, 3D modeling, and strategy games, can be slow enough even on modern desktop hardware that the user experience is disrupted by long wait times. This paper presents the Utility Coprocessor (UCop), a system that dramatically speeds up desktop applications that are CPU-bound and parallelizable by supplementing them with the power of a large datacenter compute cluster. We demonstrate several applications and workloads that are changed in kind by UCop: slow jobs that take several minutes without UCop become interactive (15–20 seconds) with it. Thanks to the recent emergence of utility-computing services like Amazon EC2 [8] and FlexiScale [45], which rent computers by the hour on a moment's notice, anyone with a credit card and $10 can use UCop to speed up his own parallel applications.

One way to describe UCop is that it effectively converts application *software* into a scalable cloud *service* targeted at exactly one user. This goal entails five requirements. *Configuration transparency* means the service matches the user's application, library, and configuration state. *Non-invasive installation* means UCop works with a user's existing file system and application configuration. *Application generality* means a developer can easily apply the system to any of a variety of applications, and *ease of integration* means it can be done with minimal changes to the application. Finally, the system must be *performant*.

UCop achieves these goals. To guarantee that the cluster uses exactly the same inputs as a process running on the client, it exclusively uses clients' data files, application images, and library binaries; the cluster's own file system is not visible to clients. The application extension is a simple user-mode library that can be installed easily and non-invasively. We demonstrate UCop's generality by applying it to the diverse application areas of 3D modeling, strategy games, and video editing; we also describe six other suitable application classes. Finally, UCop is easy to integrate: with our 295-line patch to a video editor, users can exploit a 32-node cluster (at $7/hour), transforming three-minute batch workflow for video compositing into 15-second interactive WYSIWYG display.

The biggest challenge in splitting computation between the desktop and the cloud is achieving good performance despite the high-latency, low-bandwidth network that separates them. It is dealing with this challenge that most distinguishes our work from past "department clusters," such as NOW [9], MOSIX [12], and Condor [40], which assume users and compute resources are colocated and connected by a fast network. UCop combines a variety of old and new techniques to address networking issues. To reduce latency penalties, we carefully relax the file consistency contract and use automatic profiling to send cache validation information to the server before it is needed. To reduce bandwidth penalties, we use remote differential compression. A library-level multiplexer on the cluster end of the link scales the effects of these techniques across many servers. This combination reduces UCop's overhead for remotely running a process (assuming most of its dependencies are cached in the cloud) down to just a few seconds, even on links with latencies of several hundred milliseconds.

Of course, it makes little sense to pay a remote-execution overhead of a few seconds for a computation that could be done locally in less time. UCop is also not practical for tasks that are I/O bound, or for multi-threaded applications with fine-grained parallelism. In other words, UCop will not speed up an Emacs session or reduce the wait while Outlook indexes incoming email. However, there is an important class of desktop applications that are both CPU-bound and parallelizable; UCop enhances such applications less invasively and at more interactive timescales than existing systems.

The contributions of this paper are:

- We identify a new cluster computing configuration: remote parallelization for interactive performance, which provides practical benefit to independent, individual users.
- We identify the primary challenges of this new configuration: the latency and bandwidth constraints of the user's access link.
- We introduce *prethrowing* and *task-end-to-start consistency* as techniques for dealing with that link.
- We add remote differential compression, a cluster-side multiplexer, and a shared cache, resulting in a system that can invoke a wide parallel computation using just four round-trip latencies and minimal bandwidth.
- We show our system is performant, easy to deploy, and can readily adapt existing programs into parallel services running in the cloud.

We begin with a review of related work in §2. §3 describes UCop's architecture and implementation, and §4 describes UCop applications. §5 has several evaluations: microbenchmarks (§5.1), end-to-end application benchmarks (§5.2), a decomposition of each optimization's effect (§5.3), a sensitivity analysis to latency and bandwidth (§5.4), and an analysis of the optimized system's time budget (§5.5). Finally, §6 concludes.

2 Prior Work

UCop bears similarity to prior research on computational clusters, grids, process migration, network file systems, and parallel programming models.

Computational clusters are collections of computers that are typically homogeneously configured, geographically close, and either moderately or very tightly coupled. Sprite [32] is a distributed operating system that provides a network file system, process-migration facilities, and a single system image to a cluster of workstations. MOSIX [12] is a management system that runs on clusters of x86-based Linux computers; it supports high-performance computing for both batch and interactive processes via automatic resource discovery and dynamic workload distribution. Condor [40] is a software framework that runs on Linux, Unix, Mac OS X, FreeBSD, and Windows, and supports the parallel execution of tasks on tightly coupled clusters or idle desktop machines. The Berkeley NOW [9] system is a distributed supercomputer running on a set of extremely tightly coupled workstations interconnected via Myrinet. Cluster systems have been applied to interactive applications, including some of those we consider in §4, such as compilation [28] and graphics rendering [25]. However, for transparent parallelization, clusters require the client to be one of the machines in the cluster, requiring invasive installation. By contrast, in the UCop system architecture, the client is arbitrarily configured, geographically remote, and largely decoupled from the cluster.

Computational grids [19] are collections of computers that are loosely coupled, heterogeneously configured, and geographically dispersed. Grid systems comprise a large body of work, encompassing various projects (e.g., the Open Science Grid [20] and EGEE [3]), standards (e.g., WSRF [11]), recommendations (e.g., OGSA [20]), and toolkits (e.g., Globus [18] and gLite [5]). Although the majority of work on grid systems is focused on batch processing, there has been some limited research into adding interactivity to grid systems. IC2D [14] is a graphical environment for monitoring and steering applications that employ the ProActive Java library. I-GASP [13] is a system that provides grid interactivity via a remote shell and desktop. It also includes middleware for matching applications to their required resources, which is considered by some [38] to be a critical problem for satisfying the quality-of-service requirements of interactive applications. The DISCOVER [27] system provides system-integration middleware and an application-control network to support runtime monitoring and steering of batch applications. The Interactive European Grid project [6] provides many services intended to support interactivity, including a migrating desktop, complex visualization services, job scheduling, and security services [34]. However, none of this work supports interactive applications per se, but rather provides mechanisms for interactively monitoring and manipulating a long-running distributed computation.

For a few specific types of applications, there exist massively parallel dedicated services that achieve interactive responsiveness to geographically remote, decoupled client machines. Amazon's Dynamo system employs hundreds of machines to provide real-time response to e-commerce transactions initiated by clients [16]. Google and other search engines perform brief bursts of highly parallel computation to answer clients' search queries [15]. SABRE and other online reservation systems provide clients with near-instant searching and booking for travel options [10]. However, these specialized services do not support arbitrary parallel applications, nor do they support applications whose authoritative state resides on the client machine.

Research on **process migration** is extensive; several surveys of this extensive body of work have been published [29, 31, 37]. To our knowledge, no prior work combines mechanisms and techniques as UCop does, and none of it achieves the same set of benefits. Moreover, the prior systems that are architecturally closest to UCop are not process-migration systems but network file systems. In a sense, UCop is a network file system in which the user's machine is the file server, tuned for a specific usage scenario.

Sun's NFS [36] is a basic **network file system**; in the UCop context, NFS's chatty protocol would make highly inefficient use of the high-latency connection between the client and the datacenter. The Andrew File System (AFS) [23] and Coda [26] avoid chattiness by employing leases [22]. However, leases require the ability to inspect the effect of every file system operation, which would greatly impinge on our goal of non-invasive installation; rather than simply installing a new applications, users would have to start using a new file system. UCop's prethrowing (§3.3) achieves the same performance benefits as leases without modifying the underlying file system.

The Low Bandwidth File System (LBFS) [30] is specifically aimed at improving performance over low-bandwidth WAN links. It employs caching, differential compression, and stream compression, in much the same manner as UCop does to minimize bandwidth usage (§3.4). As a general remote file system, LBFS lacks crucial optimizations for the UCop context, including our task-based consistency model, coalescing of tasks into jobs, cache sharing, and a library interface, all of which we show to be critical to achieving interactive performance (§5.3). In addition, LBFS uses leases instead of prethrowing, so the machine that holds the authoritative files (the server in LBFS terms, but the client in UCop terms) must store its files using the Arla [44] AFS client. This would require invasive installation in our scenario.

Involved **parallel programming models**, such as the Parallel Virtual Machine (PVM) [39] and the Message Passing Interface (MPI) [17], serve more tightly-coupled parallel applications. However, these are mechanisms for writing new applications. UCop's simple model, while less general, offers much easier integration for existing applications, even those not designed to exploit a cluster.

3 The Utility Coprocessor

In this section, we describe the design and implementation of the Utility Coprocessor. Sections 3.1 and 3.2 describe the programming model, and outline our implementation of its execution environment. We then describe the optimizations required to achieve good performance over a high-latency, low-bandwidth network link.

3.1 Programming model

We had several goals in designing UCop's programming model: simplicity for developers, generality across applications and operating system configurations, and good performance over slow links.

One of the mechanisms UCop uses to achieve these goals is *location independence*: applications can launch remote processes, each of which has the same effect as if it were a local process. Suppose an application's work can be divided among a set of child processes, each of which communicates with the parent through the file system or standard I/O. UCop provides a command-line utility, `remrun`, that looks like a local worker process, but is actually a proxy for a remotely running process. A simple change from `exec("program -arg")` to `exec("remrun program -arg")` provides the same semantics to the application while offloading the compute burden from the client CPU.

The consistency contract is simple: each child process is guaranteed to see any changes committed to the client file system before the child was launched, and any changes it makes will be committed back to the file system before the child terminates. Thus, dependencies among sequential children, or dependencies from child to parent, are correctly preserved. We refer to this contract as *task-end-to-start consistency* semantics. Because this contract applies to the entire file system, remote processes see *all* the same files as local client processes, including the application image, shared library binaries, system-wide configuration files, and user data.

When `remrun` is used to launch a proxy child process, it transmits an `exec` message to the cluster that includes remrun's command line arguments and environment variables. The cluster picks a worker node and launches a worker process with the specified arguments and a replicated set of environment variables, `chroot`ed into a private namespace managed by the UCop daemon (via the FUSE user-space file system framework [4]). On each read access to an existing file, UCop faults the file contents from the client; on each write to a non-existing file name, UCop creates the file in a buffer local to the node's file system. To prevent violations of task-end-to-start semantics from failing silently, UCop disallows writes to existing files. Standard input and output are shuttled between the client proxy process and the cluster worker process. When the worker process exits, UCop sends any surviving created files to the client. It also sends the process exit status; the client proxy process `exit`s with the same status.

An example best illustrates how UCop provides location independence. When compiling a single source file, UCop's `remrun gcc hello.c` produces an output file identical to a locally run `gcc hello.c`, because the remote version

- has the same `$PATH` as the client, and sees the same directories, so uses the same `gcc`;
- sees the same environment, including `$LD_LIBRARY_PATH` (shared library search path) and `$LANG` (localization);
- runs `gcc` in the same working directory, and thus finds the correct `hello.c`;
- finds the same compiler configuration and system include files; and
- writes all its output to the the client file system in the same place as if it had run locally.

Contrast this approach to other remote execution systems. Application-specific clusters such as compile and render clusters [33, 35] must be configured with a version of the compiler or renderer that matches that on the client. Grid and utility computing clusters standardize on a configuration, requiring the client configuration to conform. Process migration systems such as NOW, Condor, and MOSIX assume that user and worker machines have a network-shared `/home` and identical software configurations—for example, so that a dynamically linked executable built on a user's machine can find its shared libraries when it executes on the cluster.

The Utility Coprocessor is meant to be used by disparate independent users. No single configuration is ideal; various users sharing a cluster may have conflicting configurations. The semantics presented here hide these conflicts. Each UCop worker process mimics the client computer, and a single cluster may do so simultaneously across users and applications. As we show in §5.1.2, different Linux distributions can transparently use the same UCop cluster without any explicit pre-configuration. Our UCop cluster, which happens to use GNU libc 2.3.6, never exposes its own libraries to client applications. We have demonstrated applications that expect glibc versions as old as 2.3 and as new as 2.9.

3.2 Limitations on location independence

UCop's location-independent compute model does have limits: it extends only to the file system, environment variables, process arguments, and standard I/O pipes. Programmers UCopifying an application need to be aware of these limits. As we will show, these limits are not stumbling blocks in practice; a variety of applications can be UCopified easily.

Because UCop supports no interprocess communication other than standard I/O pipes, it precludes tightly-coupled computations in which concurrent child processes synchronize with each other using shared memory, signals, or named pipes. Some of the applications we adapted to UCop used an unsupported mechanism; our modifications primarily involved rerouting this communication through the file system (see §4).

Another limitation is that the kernel seen by a remote process is that of the cluster's worker machine, not the user's client. This is significant for two reasons. First, the semantics of system calls change slightly between kernel versions. Our application tests have not yet revealed any failures due to such a kernel incompatibility, but they are likely in code that is tightly coupled with the kernel. Second, there will be detectable differences in the machine-local state exposed by the kernel, such as process lists and socket state. UCop hides most of `/dev` and `/proc` from workers, exposing only commonly-used pseudo-devices such as `/dev/null` and `/proc/self`; the latter supports commonly-used idioms for finding loadable modules using a path relative to the currently executing image's path.

Finally, regular files on the client machine appear on the remote machine as symbolic links to files named by a hash of their contents. This is a workaround for a performance problem discussed in §5.1.3. It has little effect on most programs in practice.

3.3 Minimizing round trips

At this point, we have a basic system model with semantics suitable for the class of applications we aim to support. However, a straightforward implementation would perform poorly on a high-latency, low-bandwidth

link. We turn now to the problem of using that link efficiently by minimizing round-trip and bandwidth costs, starting with the former.

An obvious requirement for reasonable performance is to cache file contents near the cluster. The classic question is how to ensure that cached content is fresh. Neither frequent validation (à la NFS [36]) nor leases (à la AFS [23] or Coda [26]) are compatible with UCop's requirements, as detailed in §2.

Prethrow. Consistency semantics require that, for each path a worker touches during its run, we communicate the mutable binding from path to file attributes and content. A naïve implementation might ask the client for each binding on-demand, requiring one round-trip per file. We observe almost all paths touched by an application are libraries and configuration data touched on *every* run, making them easy to predict. Rather than wait for workers to request path information serially, the client sends a batch of path information likely to be useful before execution starts. We call this a *prethrow*—like a prefetch, but initiated by the sender. A prethrow is a hint: it can improve performance but does not change semantics if the prediction is wrong.

The client maintains sets of accessed paths, indexed by the first argument to `exec`. This way, the set for the 3D modeling program is maintained separately from that for the video editor. UCop prethrows only those paths that have been accessed more than once, to prevent pollution of the prethrow list by temporary files from previous runs. Currently, paths do not expire out of the prethrow list; in future versions, the server will provide a list of useless prethrows to the client after execution, to help the client decide which paths should expire.

One potential limitation of indexing by executable name is that UCop does not distinguish between two different programs invoked via the same interpreter (e.g., Python). UCop may therefore send information about paths that are not relevant. Because prethrows are hints, compact, and cachable (§3.5), this has not been a problem in practice.

3.4 Minimizing bandwidth

In a bandwidth-constrained environment, caching is critical. We adopt the well-known approach of caching by immutable hash, so that if a block of data is referred to by multiple names we only have to transmit it once.

Remote differential compression. Whole-file caching works well for files that change rarely, such as application binaries. However the user's input often changes slightly between UCop invocations. For example, a video editor's edit decision list (EDL) is a compact representation of the user's proposed manipulations to a set of (unchanging) video input files. The EDL changes

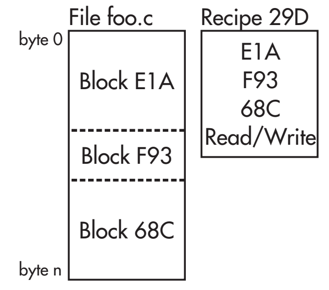


Figure 1: Objects in UCop's file synchronization protocol. To make the illustration compact, 20-byte SHA-1 hashes are represented by three hexadecimal digits.

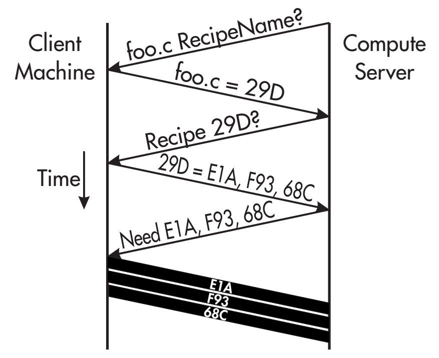


Figure 2: File transfer protocol with cold caches. To make the illustration compact, 20-byte SHA-1 hashes are represented by three hexadecimal digits.

slowly, at keyboard and mouse bit rates. *Remote differential compression* (RDC), used by LBFS [30] and rsync [42], is useful in this scenario. RDC detects which parts of a file are already cached and transmits only a small region around each changed part.

To understand our use of RDC, we first introduce some terminology (Figure 1). UCop uses the rsync fingerprint algorithm to divide all files' contents into blocks with offset-insensitive boundaries. (We plan future support for LBFS, which is more robust to modifications.) It then constructs a *recipe* for each file: a list of its constituent blocks' hashes, plus the file's permissions and ownership fields. This recipe can itself be large, so we often compactly refer to it by its hash, called a *RecipeName*.

UCop rolls whole-file caching and RDC into a single mechanism (Figure 2). Worker nodes resolve each application-requested path to a RecipeName, first by checking prethrows, then making a request to the client. If the worker recognizes the RecipeName, it knows that it already has the whole file cached. Otherwise, it requests the recipe, then any blocks from that recipe it lacks.

Stream compression. After RDC, the number of bytes that still must be transmitted can often be reduced using conventional compression. UCop compresses its channels with `zlib`.

Cache sharing. Multiple worker processes virtually always share files, such as the C library. It is wasteful for the client to send this data to each worker over the bottleneck link. Cluster nodes are interconnected with a high bandwidth network, so it is better for the client to send each file once to a cache shared by all workers.

We implemented this scheme by introducing a distributor node, called *remdis*. Remdis has a pass-through interface: it accepts jobs from the client as if it were a (fast) cluster node, and submits jobs to workers as if it were a client. Remdis forwards most messages in both directions without modification. However, it intercepts file system requests, interposing its own cache. Duplicate requests for the same content are suppressed, ensuring that no unique block is sent over the bottleneck link more than once. The Remdis cache does not change consistency semantics because RecipeNames and content block hashes describe immutable content.

In our experiments, remdis began to become a bottleneck at around 64–128 nodes. Because of its simple interface, however, it would be straightforward to build a 32-wide tree of remdis nodes to extend the distribution function to higher scales.

Job consistency. Computing and sending a prethrow message requires the client to look up the modification times of hundreds of files and transmit a few tens of KiB across the bottleneck link. Invoking n tasks incurs these costs n times. On the other hand, using the same prethrow message for all tasks, sent once and rebroadcast by remdis, reduces the prethrow cost by a factor of n.

Of course, reusing a single prethrow message violates our consistency model. If a file changes between when Task A and Task B are launched, but B uses A's prethrow message, B will see the file used by A, which is now stale. This is not a problem for groups of tasks that have no interdependencies; we call such a collection of tasks a *job*. UCop has support for *job-end-to-start consistency* semantics: each task sees any changes committed to the client file system before its enclosing *job* was launched. Applications that can operate with these semantics group tasks into jobs and generate one prethrow for each job. Other than `make`, all of the applications we deployed bundle their tasks into a job.

3.5 Client-side optimizations

The following two optimizations are performed on the client and thus involve no changes to the protocol.

Recipe caching. Constructing a recipe on the client is fast. However, some applications require hundreds of recipes, and the client can not generate a prethrow until it has them all. Thus, the client caches recipes, along with the last-modified time (mtime) of the underlying file. When a recipe is needed, the client uses the cached version if the file's mtime has not changed. For one application, this optimization saves the client from hashing 93 MiB of content, saving seconds of computation (see §5.3).

Thread interface. The `remrun` command-line utility lets applications divide their work in a natural way, creating what seem to be local worker processes but are actually proxies for remote processes. This elegance makes it trivial to expose remote execution opportunities in systems like `make`. However, simply launching 32 or 64 local processes can take several seconds, particularly on low-end desktop machines. This can consume a significant fraction of our budget for interactive responsiveness.

Thus, we added a `remrun()` library interface. A client that wants n remote processes can spawn n threads and call `remrun()` from each; the semantics are identical to spawning instances of the command-line version. The library obviates the need for extra local processes in exchange for a slightly more invasive change to the application.

3.6 Summary

In the common case, UCop incurs four round trips: the necessary one, plus three more to fault in changed user input. (We believe it is possible to eliminate all but a single RTT; see §5.5.2.) UCop also uses bandwidth sparingly. It uploads only one copy of the path attributes required by our consistency model, the per-task parameters with duplication compressed away, and the changed part of the job input. It downloads only the output data and the exit codes of the tasks.

Together, these optimizations compose an algorithm that attaches to application code with a simple interface, yet minimizes both round trips and bandwidth on the high-latency, low-bandwidth link. UCop effectively transforms *software* not originally designed for distributed computation into an efficient, highly parallel application *service* targeted at a single user.

4 Applications

In this section, we describe various classes of applications that work well with UCop. We first describe four applications we have already ported (with performance evaluations to come in §5.2). We then describe other suitable application categories.

4.1 Make

The process of adapting software to UCop is best explained by starting with a simple example: `make`, the automatic software build tool. The user's rules in the `Makefile` file tell `make` how to transform input files into output files, e.g., by invoking `gcc`. `make` assumes that when a command completes, the output file has been generated, and it is safe to launch a new command that depends on that output. That is, `make` assumes task-end-to-start consistency. Therefore, one can replace `gcc` with `remrun gcc`—literally, in the `Makefile` definitions—to push each compilation out to UCop. Additionally, `make` has a built-in facility for exploiting parallelism intended to exploit a local multiprocessor: `make -j 20` launches up to 20 concurrent processes that have no mutual ordering constraints. With `remrun`, those concurrent compiles are delegated to the UCop cluster.

Adapting `make` to UCop is trivial since it was designed to expose parallel work as separate processes communicating through the file system. For the monolithic applications we describe next, minor modification is required to expose concurrency as separate processes.

4.2 Blender

Blender [1] is a 3D modeling, animation, and rendering program written in C and C++. A common usage mode is to interactively build a model using a real-time wire-frame or shaded model; then, to refine details and lighting, the user requests a ray-traced photorealistic rendering. Since ray tracing is embarrassingly parallel, Blender has a built-in facility for exploiting local multiprocessing. Specifically, it can be easily configured to render different tiles of a scene in different threads, each accessing the current world model via shared memory.

Blender also includes a notion of a render cluster that can batch-process an animation. To use it, the user must configure a cluster with software matching her current version of Blender, with a network-mounted shared file system such as NFS, accessible over a high-bandwidth and low-latency network.

UCop can transform Blender's minutes-long batch-style frame render into an interactive-speed preview. We modified Blender's preview code to write the current 3D model to a temporary file, split the frame into very small (8-pixel-wide) tiles and dispatch a random subset of tiles to each worker node. The randomization reduces the inefficiency introduced by inter-task variance; without it, worker processes responsible for complex portions of the scene don't finish rendering until long after other workers have gone idle. As each worker completes, it writes its JPEG output tiles back to the file system. The parent UI process `waits` for the children; as each returns, the tiles are read and displayed, generating a preview that is gradually completed. These changes comprise 167 statements.

One unfortunate property of Blender is its unstable model file format: The temporary model file differs substantially from one render to the next, even when the model is unchanged. Therefore, even minor changes to the viewpoint or model require relatively large updates. UCop's remote differential compression (§3.4) is able to eliminate all but 11% of the differences. With warm caches, render requests typically transmit 779 KiB of blocks to express the input delta. With further effort, Blender might be updated to use a stable file format.

4.3 Chinese Checkers

Another suitable application class is turn-based strategy games played against a computer opponent. The effective skill of the computer is tightly linked to the amount of processing time available, making players choose between a good artificial opponent or a fast one. Interactivity is key here: it is not fun to play against an opponent that takes a dozen minutes to make each move. Traversal of AI search trees is highly parallelizable, so these games are good candidates for adaptation to UCop.

The application we use to demonstrate this class is `b1cc`, a Java program that plays Chinese Checkers [21]. Its "expert player" mode is based on a 4-deep alpha-beta-pruned minimax move-tree search.

We modified the tree search to emit a snapshot of the game configuration using the save-game function, then dispatch each branch of the first level of the search tree to a separate process. Each process computes an $(n-1)$-level alpha-beta minimax. This sacrifices our ability to prune across trees, but we expect to make up for this with the high parallelism UCop can bring to bear.

This approach was easy, but it exposes varying degrees of parallelism, and its tasks exhibit high variance. A better approach might be to locally evaluate the tree to a greater depth to find low-variance task subsets.

4.4 Cinelerra

Cinelerra [2] is an open-source video editing tool. With it, the user creates an *edit decision list (EDL)*, a metadata document that describes how source video is to be clipped, transformed, and composited into the output video. Cinelerra then performs these operations.

To test Cinelerra, we constructed a 45-second video montage composed of 29 MiB of low-resolution clips from a digicam. This montage uses only simple animated transformations, but renders 8.3× slower than realtime. Cinelerra offers many compute-intensive effect plugins that slow down previewing even more.

Like Blender, Cinelerra includes a notion of a render cluster that depends on explicit version configuration and a fast network. Its "background render" function breaks a clip into frames and pre-renders the sequence of frames so the operator can preview the sequence at full frame rate. We modified Cinelerra to emit a set of control files (in Cinelerra's native job-control language) that divide the preview region temporally into brief snippets. It launches one child process to render each snippet to MPEG, then collects the MPEGs into a render timeline and plays them in order.

The biggest constraint on using Cinelerra with UCop is getting enormous video inputs to the cluster. Our 29-MiB input videos represent amateur video editing; serious editing will use multi-GiB input files. While they are read-only and thus their size does not affect a warm-cache scenario, big inputs produce substantial transmission delay in a cold-cache scenario.

Three techniques may mitigate this constraint. First, UCop might demand-fault individual blocks rather than entire files; this can help if only small portions of the input videos are used in the output. Second, Cinelerra might transcode video at the client into lower-quality drafts to exploit UCop even when transmission delays are dominant. Third, a user might fault in media to a UCop cluster (e.g., by running `remrun md5sum movie.avi` on it) the day before sitting down to edit.

4.5 Other applications

Beyond the applications we have modified, many other application classes can exploit UCop. The best applications are those where small changes to input incur CPU-bound and coarsely parallelizable computation. This section has some examples.

One potential class is mathematics software. For instance, numeric modeling packages such as Matlab and Octave parallelize vector math, and symbolic math packages such as Macsyma and Mathematica parallelize manipulation of independent subexpressions. Also, spreadsheet applications have parallelizable data-flow models.

Speech dictation software often performs a great deal of processing to parse a small amount of user speech. A researcher familiar with the area claims desktop access to parallel resources would improve quality and enable new applications [43].

Interactive GIS applications often perform CPU-intensive tasks, such as rendering a large database of vector data into a bitmap or performing convolutions on large bitmaps (e.g., reprojecting maps or aerial photographs). In these applications, small user inputs such as changes in view or layer registration can change the global configuration and necessitate CPU-bound re-rendering.

Photo manipulation software may also be readily adaptable to UCop. Photoshop and GIMP are adopting a nondestructive editing model, i.e., recording a stack of operations rather than just their cumulative effects. This stack is essentially an edit decision list, which UCop could send concisely. Image filters are both coarsely parallelizable and slow, making them a good fit for UCop.

Finally, software analysis tools, such as model checkers, whole-program static analyzers, and theorem provers generate substantial parallel workloads and are often used as part of a developer's interactive workflow.

Note that like Blender and Cinelerra, some of these applications already have support for a single-purpose, locally-administered, tightly-coupled cluster. Some even sell dedicated cluster hardware [7]. UCop, in contrast, is general: a single cluster running a single piece of software that can service all these applications simultaneously.

5 Evaluation

Our evaluation of UCop is divided into five parts. We begin with microbenchmarks in §5.1. End-to-end application benchmarks are described in §5.2. In §5.3, we analyze the efficacy of UCop's protocol optimizations, showing how performance suffers as each is disabled. We present a sensitivity analysis to latency and bandwidth in §5.4. Finally, in §5.5, we decompose how a typical UCop task spends its time budget.

All of our experimental clusters were constructed from Amazon's EC2 "Elastic Compute Cloud" service. Each VM is one of Amazon's "high-CPU medium" instances: a Xen virtual machine with 1.7 GB of memory and 2 CPU cores, each of which is approximately equal to a 2.5GHz Opteron or Xeon processor, circa 2007. Within EC2, we measured a typical interconnect bandwidth of 800 Mbit/sec and RTT of 600 μsec. As we will see in §5.2, most tests used artificial bandwidth and latency restrictions to emulate the typical case of a client separated from the compute cluster by a bottleneck link.

5.1 Microbenchmarks

5.1.1 Correctness

Our task-close-to-open consistency model and whole-file-system replication scheme were designed to let remote processes produce results identical to those produced by local computation. To verify this property, we used UCop to build GNU Coreutils v7.1, a collection of 102 system utilities. The build process has an intricate dependency structure and invokes hundreds of sub-tasks, many of which redirect `stdin` or `stdout` to other programs or the file system. Errors in UCop's consistency model or its implementation are likely to cause build failures (and did so in early versions of UCop). Adapting the build process to UCop only required typing

OS Distribution	libc	gcc	kernel
Centos 5.2	2.5	4.1.2	2.6.18
Debian 3.1	2.3.2	3.3.5	2.6.16
Debian 4.0	2.3.6	4.1.2	2.6.21.7
Debian 5.0	2.7	4.3.2	2.6.21.7
Debian 6.0 beta	2.7	4.3.3	2.6.21.7
Fedora Core 8	2.7	4.1.2	2.6.21.7
Gentoo 2008.0	2.6.1	4.1.2	2.6.18
Ubuntu 9.04	2.9	4.3.3	2.6.21.7

Table 1: Linux distributions used as clients to compile GNU Coreutils with a UCop cluster running Debian 4.0. In each case, UCop generated binaries identical to those compiled locally.

`./configure CC="remrun gcc"`. We also compiled Coreutils locally; all the locally-compiled outputs were identical to their cluster-compiled counterparts.

Further supporting our claim of location independence is our (accidental) discovery that `remrun` is "self-hosting." An author was tinkering with remrun commands when he noticed the system had slowed, and was transferring files that seemed unrelated to his task. He eventually discovered that he'd accidentally edited his command-line to read `remrun remrun gcc hello-world.c`—that is, a recursive call to remrun. The command still ran correctly; the remrun client ran on the server automatically.

We also built this paper using `remrun latex paper.tex`. The emitted .dvi file differed from a locally-built copy in one byte: the minute field of a timestamp.

5.1.2 Configuration Transparency

To test UCop's insensitivity to heterogeneous clients, we repeated the Coreutils build test on various distributions of Linux, only one of which matched the version running on the UCop cluster itself (Debian 4.0). Each distribution generated distinct outputs due to variations in the versions of gcc and libc. Indeed, simply invoking Debian 5.0's `gcc` binary on a Debian 4.0 machine fails due to shared library incompatibility. Using UCop, however, every locally-built binary matched the binary the cluster built on that client's behalf. The distributions we tested are shown in Table 1.

5.1.3 FUSE Performance

We implemented UCop's on-demand file system using FUSE [4], a user-space file system framework. This simplified development, but proxying every file system operation through user-space has a significant performance cost for I/O intensive processes. In this section, we evaluate a technique to mitigate this cost.

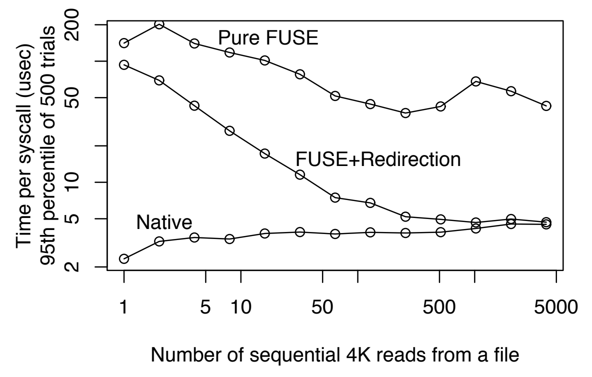


Figure 3: Log-log plot of the amortized time per syscall after one `open`, n `reads`, and one `close` on a file. All reads are 4,096 bytes. The file was resident in the buffer cache.

When a worker process tries to open a path corresponding to an extant file on the client, FUSE instantiates that path not with a local file but with a symbolic link to a file with the appropriate contents. This file is kept on a kernel-managed native file system. Thus, access to extant files is mediated by FUSE only during `open`; other operations like `read` are handled much more efficiently.

We quantified the advantage of our approach by measuring the time required to open, read, and close a file. Figure 3 shows the amortized cost per syscall for sequential 4,096 byte `reads` with a warm buffer cache, plotted for a range of `reads` per `open`.

The top curve shows the performance of a FUSE-managed file system without redirection. The bottom curve shows the performance of Linux's native ext3 file system. The slowdown is significant, and is worst for small numbers of reads, ranging from 10× to 60×. The middle curve shows amortized read performance with our symlink scheme in place. The optimization never hurts performance, and after about 40 `reads`, improves amortized performance to within 2× native.

Note that the optimization does slightly hurt transparency: all files seen by the workers are symbolic links.

5.2 Application Benchmarks

In this section, we describe end-to-end benchmarks for three applications run on UCop, tested under realistic network conditions. In most tests, the client and cluster were both within Amazon EC2, with latency artificially injected and bandwidth constrained by Linux Traffic Control [24]. The exception is Section 5.2.5, in which we ran experiments from a real coffee shop.

To determine what network conditions should be emulated for our EC2-based experiments, we tested the networks at various locations in Seattle that offer public wireless Internet access. These included two coffee

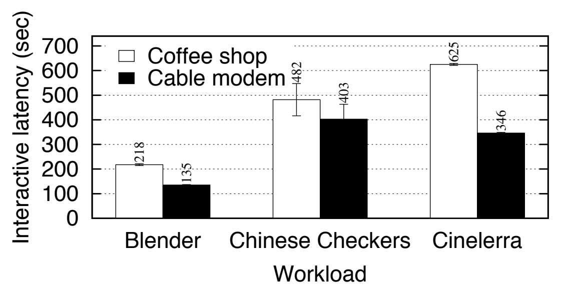


Figure 4: Run times with cold caches and 32 workers. Error bars show 95% confidence interval around displayed mean.

shops, a cell phone company hot-spot, and a restaurant. At each, we characterized the access-link bandwidth and latency to EC2 (which is across the country, in Virginia). Average latency ranged from 121 ms for the cell phone company to 199 ms for one of the coffee shops. Upstream bandwidth occupied a fairly narrow range: from 1300 Kib/s for one of the coffee shops to 1500 Kib/s for the restaurant. Downstream bandwidth also occupied a narrow range: from 1400 Kib/s for the same coffee shop to 1600 Kib/s for the restaurant.

In the experiments that follow, the *coffee-shop* configuration models a round-trip time of 160 ms, an upstream available bandwidth of 1400 Kib/s, and a downstream available bandwidth of 1500 Kib/s. A second *cable-modem* configuration, based on the authors' home offices, models 70 ms RTT, 4 Mib/s upstream, and 16 Mib/s downstream.

In most experiments, the local-computation measurements are run on exactly the same kind of machine as the cluster worker nodes. The exception is §5.2.5, where, out of necessity, the client was a laptop.

5.2.1 Cold cache performance

When caches are cold, UCop is slow. Files are faulted in serially over the bottleneck link, necessitating at least as many RTTs and transfer delays are there are files. Figure 4 shows that warm-up times are from 2 to over 10 minutes. Cinelerra is slowest, with round trips proportional to its 484 paths, and bandwidth costs proportional to its 93 MiB of files.

The evaluations that follow all measure performance once the caches are warm. In each experiment, we first destroy all cached data, then warm the caches by invoking the test application twice. Finally, we collect a data point by timing the application's performance when given a new (uncached) input for the first time: a tweaked Cinelerra edit decision list, a modified Blender model, or a new move in a game of Chinese Checkers. We repeat each experiment 10 times and plot the mean. Around each mean we show, using error bars, the 95% confidence interval for the mean using Student's t-distribution.

Cold-cache performance is slow; UCop performs poorly for applications run only once. Of course, the same can be said of client software, whose installation also typically takes several minutes. Though currently unimplemented, caches could be made persistent across cluster instances and even users; clusters would then boot ready-to-use in the common case.

5.2.2 Blender

The first application we test is Blender (§4.2). This test renders a 14.2-MiB model of the Starship Enterprise [41] at HD quality (1920×1080). Figure 5 compares the time to run locally with the time to run at various levels of parallelism in UCop.

First, observe that in the local case, rendering takes 137 sec, a duration most users would consider non-interactive and that would cause them to task-switch. Next, observe that in either of our network configurations, a cluster size of two breaks even with the local case; even a small degree of parallelism overcomes the overhead of remote operation. Finally, observe that from the coffee shop, 64 workers render the scene in 20 sec, and, using a cable modem, 64 workers take 18 sec. These results show that even on long-delay, low-bandwidth networks, UCop can perform complex rendering in seconds, turning it from a batch to an interactive operation.

5.2.3 Chinese Checkers

The next experiment measures the time it takes for the Chinese Checkers expert algorithm to make a move. As described earlier, in the local case, Chinese Checkers uses a sequential pruning tree search; for both local and remote tests, we use only one core per machine. We automate the game by driving the expert mode (with and without UCop) against a locally-executed novice opponent. We measure only the time taken to compute the expert's most complex move.

Figure 5 shows the results. Computed locally, the computer's move takes 317 sec, a long enough wait that the game might not be fun. UCop overcomes the remote-processing overhead by degree 3. With a 64-node cluster, the worst-case move time is reduced to 26 sec in the coffee shop, and to 23 sec with a cable modem. This illustrates how UCop can make strategy games enjoyably interactive even at expert levels.

5.2.4 Cinelerra

The third application is the Cinelerra video editor. Because video playback results play out over time, total completion time is not an interesting metric; therefore,

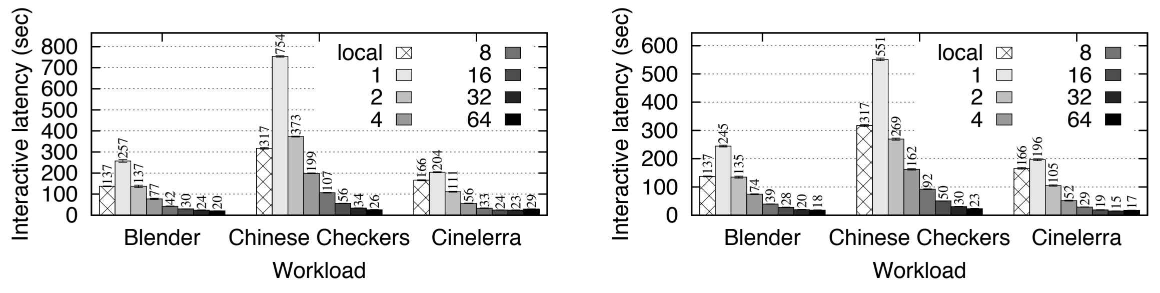


Figure 5: Run times with local computation and varying degrees of UCop parallelism; smaller times are better. Network configurations are *coffee-shop* (left) and *cable-modem* (right). The parallel cluster uses the same class of machine as the local-computation baseline. Error bars show 95% confidence interval around displayed mean.

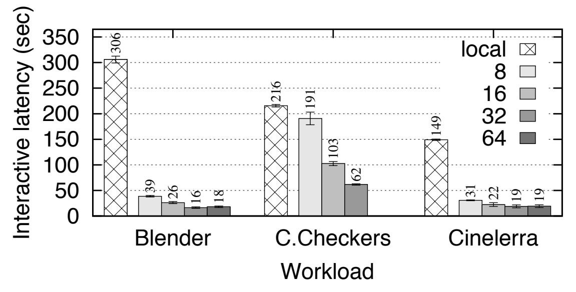


Figure 6: The effect of parallelism when the client is a laptop in a real coffee shop. Note that, unlike in Figure 5, the locally-computed runs execute on a different class of machine than is used in the compute cluster. Error bars show 95% confidence interval around displayed mean.

our measure of interactive latency for Cinelerra is *preroll time*. This is the delay until video playback could theoretically begin and still allow uninterrupted complete playback. The workload is a 20-second clip of the digicam montage described in §4.4.

Figure 5 shows the results. In the local case, the delay is 166 sec, a long time to preview a clip. UCop begins showing benefit at degree 2, readily overcoming the remote overhead. Finally, by degree 32, Cinelerra delivers the same clip in only 23 sec from a coffee shop, or 15 sec using a cable modem.

5.2.5 Tests from a real coffee shop

The previous sections reported experiments done in a controlled environment meant to emulate a coffee shop. We now discuss experiments using an *actual* coffee shop. In these tests, the client is a laptop, a Lenovo z61p with a 2GHz Core Duo running Debian Lenny. The workers are still EC2 cluster machines. Blender is linked against Debian Etch, and thus runs from inside a `kvm` hardware virtual machine, which is limited to one core. Figure 6 shows the results; they are essentially similar to, and thus validate, our earlier emulated-environment results.

5.2.6 Discussion

UCop's goal is to improve interactive performance by achieving low latency; computational *efficiency* is less important. Indeed, speedup per node in these tests peaks at 1–2 nodes (56–85%) and decreases monotonically thereafter, sinking to 12–35% for 32–64 nodes. For this reason, in the following experiments we consider only 32-node clusters, as they provide reasonably low latencies at reasonable cost.

"Reasonable" latency is difficult to define objectively. Results are highly sensitive to workload characteristics; any selection is, to some degree, arbitrary. Had we chosen simpler workloads, the desktop might have rendered them quickly, obviating the need for the remote cluster. More complex workloads favor UCop: Overheads limit UCop's ability to use a larger cluster to reduce wait time, but UCop *can* often use a larger cluster to hold wait time constant as the workload complexity increases dramatically. Similarly, due to the preroll metric, a short Cinelerra clip duration favors local computation, and longer durations favor UCop.

5.3 Decomposition

The preceding section shows that UCop is performant. We now break down the contribution of each optimization described in §3. As Figure 7 shows, each optimization is necessary for good performance, although some are more important for certain applications.

Prethrow. The first group in both graphs of Figure 7 shows that prethrowing is the most important optimization. Cinelerra, which accesses 484 paths, is most affected by the loss of prethrown attributes. Because files are validated sequentially, the worst slowdown is seen in the highest-latency configuration, *coffee-shop*.

Remote differential compression. The next group shows performance when RDC is disabled. That is, every byte of changed files must be uploaded, rather than just the changed bytes. Blender fares worst without this optimization, since its input file is the largest.

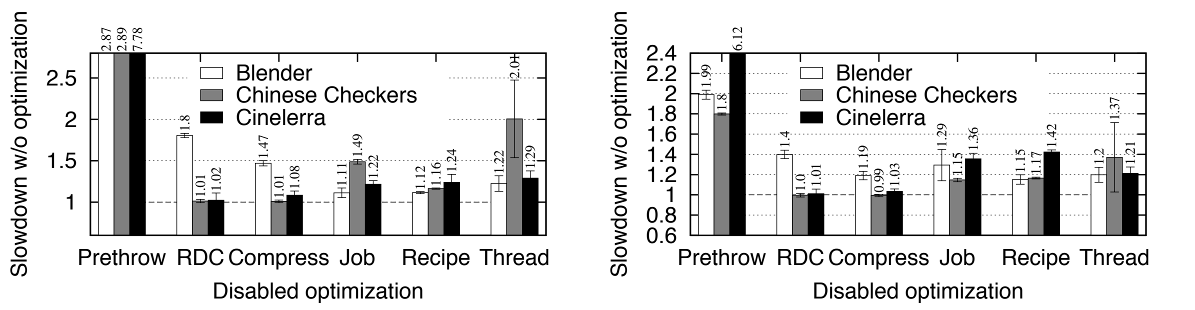


Figure 7: Run times with optimizations disabled. Values are normalized to means in Figure 5, so 1.0 means the optimization has no effect. Network configuration is *coffee-shop* (left) and *cable-modem* (right). Parallelism is 32. Error bars show 95% confidence interval around displayed mean.

Stream compression. In the next group, we show the effect of disabling stream compression. Again, this affects Blender the most, since even with RDC, Blender's unstable output requires UCop to transmit 779 KiB of changed blocks; performance suffers in this test because the blocks are compressible.

Cache sharing. Without cache sharing within the cluster, all worker nodes must get everything directly from the client via the bottleneck link. This inflates all overheads by a factor equal to the cluster size. Clearly, this is unscalable. Indeed, this experiment ran so slowly that we abandoned collecting statistically-significant data; none appear in the figures.

Job consistency. The next group of bars shows the benefit of job-end-to-start consistency. Here, no two tasks share the same job, and hence consistency semantics demand that each task prethrow its own path list. This has no effect on round-trips, but congests the link with duplicate data.

Recipe caching. In the next group of bars, the client does not cache recipes as described in §3.5; therefore, the client must compute megabytes of hashes before it can begin the transaction with a prethrow.

Thread interface. The last group shows the value of launching UCop tasks from threads rather than processes. Here, the cost is the serialized overhead on the client machine. Chinese Checkers suffers the most because it launches 86 tasks; the other applications launch 32. This optimization will be more important on slow clients (our experimental client is fast), and at higher degrees of parallelism.

5.4 Sensitivity analysis

Figures 8 and 9 show the sensitivity of our results to network characteristics. UCop is effective at dramatically reducing sensitivity to poor latency and bandwidth conditions. These experiments also show how UCop achieves this. Without prethrow, performance is highly sensitive to latency. Without compression and RDC, performance is highly sensitive to bandwidth, at least for Blender, which is bandwidth-intensive.

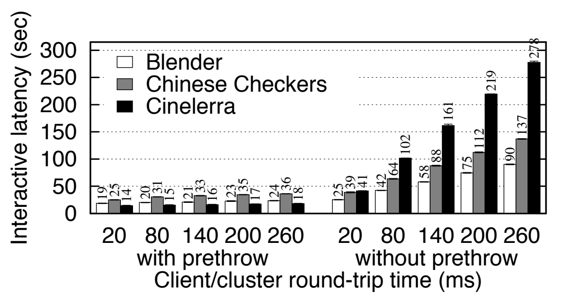


Figure 8: Interactive latencies with varying client/cluster RTT, *cable-modem* available bandwidth, and 32 worker nodes. Smaller times are better. Prethrows make UCop's performance nearly latency-insensitive. Error bars show 95% confidence interval around displayed mean.

5.5 Remaining costs

The previous section explained the techniques that achieve interactive-scale performance. This section explores the present limits to performance and how it might be further improved.

5.5.1 Latency breakdown

To guide this discussion, Figure 10 provides a rough analysis of how the 20 sec of interactive latency for our Blender workload arose. We estimated the latency components as follows:

Launch overhead. We measured `remrun` on EC2 with no artificial latency or bandwidth throttling. It spent about 1 sec launching 32 job requests, including time to `stat` the file system for each prethrow path and fault in missing blocks.

Process start. We asked Blender to load the Enterprise model, then exit without rendering anything; this took about 1.5 sec.

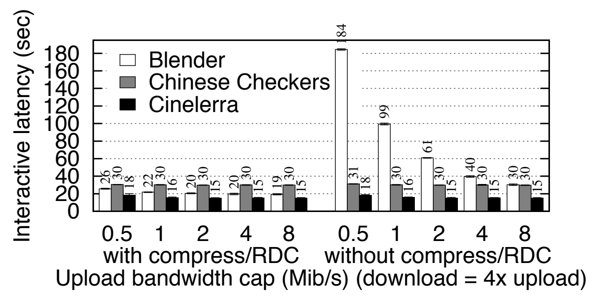


Figure 9: Interactive latencies with varying available bandwidth, *cable-modem* client/cluster RTT, and 32 worker nodes. Smaller times are better. RDC and compression make UCop's performance nearly bandwidth-insensitive. Error bars show 95% confidence interval around displayed mean.

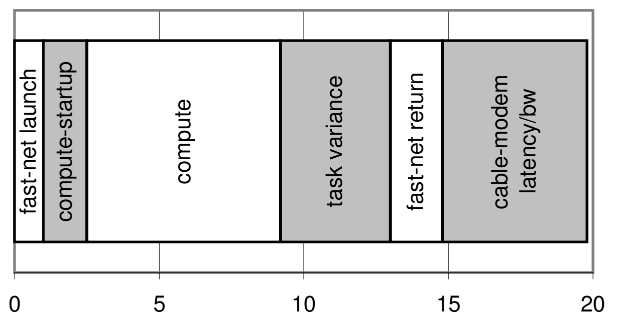


Figure 10: Approximate components of overall latency: Blender on a 32-node cluster.

Computation. Rendering one of the 32 tiles takes 6.7 sec on average, with a maximum of 10.5 sec. This variance arises because nodes rendering the blackness of space surrounding the Enterprise become idle long before the most-loaded node puts finishing touches on the glow of the warp nacelles. Thus, we account for the time as 6.7 sec of computation plus 3.8 sec of inter-task variance. Contrary to prior reports [46], we observed no significant effect from inter-*machine* variance; the time required to complete a task correlated with the task's workload, not the machine that ran it.

Results download. Our test with an unthrottled EC2 network showed that UCop spends 1.8 sec organizing and returning the result tiles.

Real network costs. The costs above account for all but 5 sec of the cable-modem time. We attribute these remaining 5 sec to the network delays due to increased RTT and reduced available bandwidth.

5.5.2 Opportunities for improvement

The firmest contributor to latency is the compute time itself. Of course, wider parallelization can help, but the benefit is constrained by inter-task variance and offset by an increase in network launch and return costs.

The four round trips we spend are three more than strictly necessary. Eliminating them as follows could save as much as 0.5 sec in the coffee shop. The prethrow technique already profiles applications to predict paths with stable contents. The same technique could detect repeatedly-used paths with consistently fresh contents, and use that cue to eliminate the RecipeNameRequest round-trip. Reasonably assuming that the RecipeName itself is fresh, the client could pipeline the recipe as well. Finally, by maintaining a shadow index of blocks the cluster already knows, the client could further pipeline the set of new blocks, eliminating a third RTT.

Process startup might be mitigated by checkpointing the client-side process after basic initialization or after parsing stable inputs. In addition, each process's compute time might be predictable in some cases; shorter subtasks might automatically be run locally, or run locally in the case of network failure.

6 Conclusions

The Utility Coprocessor is a new use for utility compute clusters: dramatically enhancing the performance of CPU-intensive, parallelizable desktop applications. UCop's non-invasive installation and automatic support for arbitrary client software configurations lets users farm their desktop compute tasks out to the cloud without changing their model of where files are stored or how new software is installed.

The primary challenge in making a system like UCop performant is overcoming the relatively high latency and low bandwidth of the link separating the user's desktop from the compute cloud. We introduce the techniques of task-end-to-start consistency and prethrowing to avoid latency penalties. We avoid bandwidth penalties using a combination of cluster-wide cache sharing, remote differential compression, and the notion of job-end-to-start consistency. We also use a variety of techniques to minimize the client's CPU and I/O load when sending work to a cluster. Taken together, these techniques allow UCop to efficiently execute wide parallel computations in the cloud with low overhead, even though all the canonical state is on the client.

Our evaluation demonstrates speedup with only 2–3 nodes even in a challenging coffee-shop network environment, and 15–20 second interactive performance with 32–64 nodes. We show the necessity of each of UCop's optimizations, and that the optimized system is insensitive to latency and bandwidth variations. We also identify further opportunities for improving performance.

The Utility Coprocessor is a novel and practical system for easily and inexpensively improving the performance of CPU-bound desktop applications. It is general to many applications, and UCop support is easy for developers to integrate. Thanks to the availability and low cost of utility computing clusters like Amazon EC2, the power of UCopified applications is available to individuals—today.

References

[1] Blender 3D Modeling Suite. http://blender.org/.

[2] Cinelerra Video Editor. http://cinelerra.org/.

[3] Enabling Grids for E-sciencE (EGEE). http://www.eu-egee.org/.

[4] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[5] gLite Lightweight Middleware for Grid Computing. http://glite.web.cern.ch/glite/.

[6] Interactive European Grid Project. http://www.i2g.eu/.

[7] Advanced Cluster Systems. Math supercomputer-in-a-box. http://www.advancedclustersystems.com/ACS/Products.html.

[8] Amazon Web Services. EC2 elastic compute cloud. http://aws.amazon.com/ec2.

[9] Anderson, T. E., Culler, D. E., Patterson, D. A., and the NOW team. A case for NOW (networks of workstations). *IEEE Micro 15*, 1 (1995), 54–64.

[10] Anthes, G. Sabre flies to open systems. *Computerworld* (May 2004).

[11] Banks, T. Web services resource framework (WSRF) primer v1.2. Tech. Rep. wsrf-primer-1.2-primer-cd-02, 2006.

[12] Barak, A., Guday, S., and Wheeler, R. G. *The MOSIX Distributed Operating System - Load Balancing for UNIX*, vol. 672 of *Lecture Notes in Computer Science*. Springer, 1993.

[13] Basu, S., Talwar, V., Agarwalla, B., and Kumar, R. Interactive grid architecture for application service providers. In *ICWS* (2003), pp. 365–374.

[14] Baude, F., Caromel, D., Huet, F., Mestre, L., and Vayssière, J. Interactive and descriptor-based deployment of object-oriented grid applications. In *HPDC* (2002), pp. 93–102.

[15] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[16] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), ACM, pp. 205–220.

[17] Forum, M. P. I. MPI: a message-passing interface standard version 2.1.

[18] Foster, I. Globus Toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing* (2006), pp. 2–13.

[19] Foster, I., and Kesselman, C. *The Grid - Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, 1999.

[20] Foster, I. *et al.* The Open Grid Services Architecture, version 1.5. Tech. Rep. GFD-I.080, 2006.

[21] Foucaud, F., Johary, R., and Terral, J. Bordeaux1 Chinese Checkers. http://sourceforge.net/projects/b1cc.

[22] Gray, C. G., and Cheriton, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989), pp. 202–210.

[23] Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., and West, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 51–81.

[24] Hubert, B. Linux advanced routing & traffic control. http://lartc.org/.

[25] Humphreys, G., Houston, M., Ng, R., Frank, R., Ahern, S., Kirchner, P. D., and Klosowski, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH* (2002), pp. 693–702.

[26] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the Coda file system. *ACM Transactions On Computer Systems 10*, 1 (Feb. 1992), 3–25. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/s13.pdf.

[27] Mann, V., and Parashar, M. DISCOVER: A computational collaboratory for interactive grid applications. In *Grid Computing: Making the Global Infrastructure Reality* (January 2003), John Wiley and Sons, pp. 727–744.

[28] McClure, S., and Wheeler, R. Mosix: How Linux clusters solve real-world problems. In *USENIX Annual Technical Conference, FREENIX Track* (2000), pp. 49–56.

[29] Milojicic, D. S., Douglis, F., Paindaveine, Y., Wheeler, R., and Zhou, S. Process migration. *ACM Computing Surveys 32*, 3 (September 2000), 241–299.

[30] Muthitacharoen, A., Chen, B., and Mazières, D. A low-bandwidth network file system. In *SOSP* (2001).

[31] Nuttall, M. Survey of systems providing process or object migration. *Operating Systems Review 28* (1994), 64–80.

[32] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., and Welch, B. B. The Sprite network operating system. *Computer 21*, 2 (1988), 23–36.

[33] Petersen, D. Loki Blender queue manager. http://sourceforge.net/projects/loki-render/.

[34] Płóciennik, M., Owsiak, M., Fernández, E., Heymann, E., Senar, M. A., Kenny, S., Coghlan, B., Stork, S., Heinzlreiter, P., Rosmanith, H., Plasencia, I. C., and Valles, R. Int.eu.grid project approach on supporting interactive applications in grid environment. In *Workshop on Distributed Cooperative Laboratories: Instrumenting the GRID (INGRID)* (April 2007).

[35] Pool, M. distcc, a fast free distributed compiler. In *Linux.conf.au* (2003). http://distcc.samba.org/doc/lca2004/distcc-lca-2004.html.

[36] Sandberg, S., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun network file system. In *USENIX Summer Conference* (June 1985), pp. 119–130.

[37] Smith, J. M. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev. 22*, 3 (1988), 28–40.

[38] Stefano, A. D., Pappalardo, G., Santoro, C., and Tramontana, E. Supporting interactive application requirements in a grid environment. In *Workshop on Distributed Cooperative Laboratories: Instrumenting the GRID (INGRID)* (April 2007).

[39] Sunderam, V. S. PVM: A framework for parallel distributed computing. In *Concurrency: Practice and Experience* (1990), vol. 2, pp. 315–339. http://www.csm.ornl.gov/pvm/.

[40] Thain, D., Tannenbaum, T., and Livny, M. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience 17*, 2-4 (2005), 323–356.

[41] Thomas, W. U.S.S. Enterprise Blender mesh, 2005. http://stblender.iindigo3d.com/meshes_startrek.html.

[42] Tridgell, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[43] Wang, K. Microsoft Research. Personal communication.

[44] Westerlund, A., and Danielsson, J. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track* (1998), USENIX.

[45] XCalibre. Flexiscale. http://www.flexiscale.com.

[46] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., and Stoica, I. Improving MapReduce performance in heterogeneous environments. In *OSDI* (2008), pp. 29–42.

Apiary: Easy-to-Use Desktop Application Fault Containment on Commodity Operating Systems

Shaya Potter Jason Nieh
Computer Science Department
Columbia University
{spotter, nieh}@cs.columbia.edu

Abstract

Desktop computers are often compromised by the interaction of untrusted data and buggy software. To address this problem, we present Apiary, a system that transparently contains application faults while retaining the usage metaphors of a traditional desktop environment. Apiary accomplishes this with three key mechanisms. It isolates applications in containers that integrate in a controlled manner at the display and file system. It introduces ephemeral containers that are quickly instantiated for single application execution, to prevent any exploit that occurs from persisting and to protect user privacy. It introduces the Virtual Layered File System to make instantiating containers fast and space efficient, and to make managing many containers no more complex than a single traditional desktop. We have implemented Apiary on Linux without any application or operating system kernel changes. Our results with real applications, known exploits, and a 24-person user study show that Apiary has modest performance overhead, is effective in limiting the damage from real vulnerabilities, and is as easy for users to use as a traditional desktop.

1 Introduction

In today's world of highly connected computers, desktop security and privacy are major issues. Desktop users interact constantly with untrusted data they receive from the Internet by visiting new web sites, downloading files, and emailing strangers. All these activities use information whose safety the user cannot verify. Data can be constructed maliciously to exploit bugs and vulnerabilities in applications, enabling attackers to take control of users' desktops. For example, a major flaw was recently discovered in Adobe Acrobat products that enables an attacker to take control of a desktop when a maliciously constructed PDF file is viewed [2].

The prevalence of untrusted data and buggy software makes application fault containment increasingly important. Many approaches have been proposed to isolate applications from one another using mechanisms such as process containers [24, 28, 32] or virtual machines [39]. Faults are confined so that if an application is compromised, only that application and the data it can access are available to an attacker.

However, existing approaches suffer from an unresolved tension between ease of use and degree of fault containment. Some approaches [20, 26] provide an integrated desktop feel but only provide partial isolation. They maintain traditional usage metaphors, but do not prevent vulnerable applications from compromising the system itself. Other approaches [34, 39] have less of an integrated desktop feel but fully isolate applications into distinct environments, typically by using separate virtual machines. These approaches effectively limit the impact of compromised applications, but are harder to use because users are forced to learn new ways to use these systems as well as having to manage many environments.

To address these problems, we introduce Apiary, a system that provides strong isolation for robust application fault containment while retaining the integrated look, feel, and ease of use of a traditional desktop environment. Apiary accomplishes this using three key mechanisms that combine well-understood technologies like thin clients, operating system containers, and unioning file systems in novel ways.

First, it decomposes a desktop's applications into isolated containers. Each container is an independent appliance that provides all services an application needs to execute. This prevents an application exploit from compromising other applications. To retain traditional desktop semantics, Apiary integrates these containers in a controlled manner at the display and file system.

Second, it introduces the concept of ephemeral containers. Ephemeral containers are execution environments with no access to the user's data that are quickly instantiated from a clean state for only a single application execution. When the application terminates, the container is archived and never used again. Ephemeral containers have three benefits. First, they prevent compromises, because exploits, even if triggered, cannot persist. Second, they protect users from compromised applications. Even when an application has been compromised, a new ephemeral container running that application in parallel will remain uncompromised. Third, they help protect user privacy when using the Internet. Apiary uses ephemeral containers as a fundamental building block of the integrated desktop experience.

Third, Apiary introduces the Virtual Layered File System (VLFS). Apiary introduces the VLFS to efficiently store and instantiate containers. Each software package or application is stored as a read-only software file system layer. Layers are analogous to software packages in current systems. A VLFS dynamically composes together a set of shared software layers into a single file system view. In Apiary, each container has its own independent VLFS. Since each container's VLFS will share the layers that are common to them, Apiary's storage requirements are the same as a traditional desktop. Similarly, since no data has to be copied to create a new VLFS instance, Apiary is able to quickly instantiate ephemeral containers for a single application execution.

We have implemented an Apiary Linux prototype without any application or operating system kernel changes. We evaluated its effectiveness by conducting various experiments with real applications, vulnerabilities, and users in a user study. Our results show that Apiary can instantiate application containers in under a second, can upgrade a set of containers in under a seconds, has scalable storage requirements, and has only modest file system performance overhead. Our results show that Apiary is effective at containing real exploits. It quickly returns the desktop to a clean uncompromised state in cases where the exploit forces a complete reinstall when it occurs on a traditional desktop system. Finally, our results from a blind user study show that users find Apiary as easy to use as a traditional desktop.

2 Apiary Usage Model

Figure 1 shows the Apiary desktop. It looks and feels like a regular desktop. Users launch programs from a menu or from within other programs, switch among launched programs using a taskbar, interact with running programs using the keyboard and mouse, and have a single display with an integrated window system and clipboard functionality that contains all running programs.

Although Apiary works similarly to a regular desktop, it provides fault containment by isolating applications into separate containers. Containers provide all the resources an application needs to run. This includes an isolated execution context, independent display driver and complete file system. As each container's file system is independent, each container has its own isolated home directory to store files created by the user in that container and to isolate them from every other container. For example, if one had a web browsing container and a word processing container, each application would store their contents in the container's version of the user's home

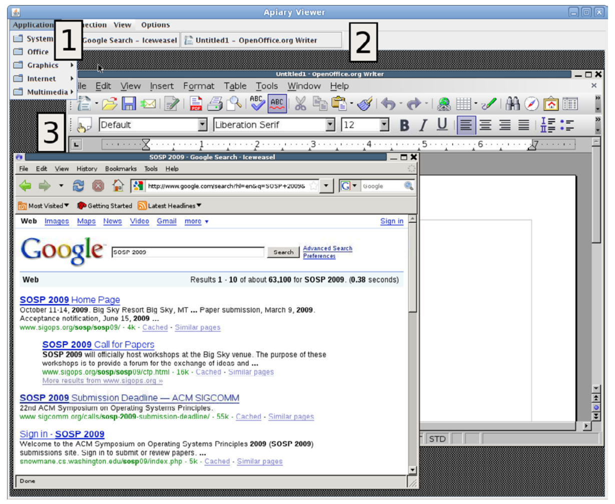


Figure 1: Apiary desktop session: (1) application menu, (2) window list, (3) composited display

directory. This enables containers to enforce isolation without the creation of any isolation rules.

Apiary isolates individual applications, not individual programs. An application in Apiary is a software appliance made up of multiple programs that are used together in a single environment to accomplish a specific task. For instance, a user's web browser and word processor are separate applications and isolated from one another. This software appliance model means that users can install separate isolated applications that contain many or all of the same programs, but used for different purposes. For example, a banking application contains a web browser for accessing a bank's website, while a web surfing application also contains a web browser, but for general web browsing. Both appliances make use of the same web browser program, but are listed as different applications in the application menu. This model can be extended to the point where individual containers are provided for many individual sites, such as Amazon and eBay. While this model differs from a regular desktop, it is similar to what users experience on mobile devices, such as iPhone and Android phones, where they install website specific applications to gain more efficient access to those sites.

Apiary provides two types of containers: ephemeral and persistent. Ephemeral containers are created fresh for each application execution. Persistent containers maintain their state across application executions. Apiary lets users select whether an application should launch within an ephemeral or a persistent container.

Ephemeral containers provide a powerful mechanism for protecting desktop security and user privacy. Users will typically run multiple ephemeral containers, even for the same application, at the same time. They provide important benefits for a wide range of uses.

Ephemeral containers prevent compromises because exploits cannot persist. For example, a malicious PDF

document that exploits an ephemeral PDF viewer will have no persistent effect on the system because the exploit is isolated in the container and will disappear when the container finishes executing.

Ephemeral containers protect user privacy when using the Internet. For example, many websites require cookies to function, but also store advertisers' cookies that can compromise a user's privacy. Apiary makes it easy to use multiple ephemeral web browser containers simultaneously, each with separate cookies, and prevents the cookies from persisting.

Ephemeral containers protect users from compromises that may have already occurred on their desktop. If a web browser has been compromised, parallel and future uses of the web browser will allow an attacker to steal sensitive information when the user accesses important websites. Ephemeral containers are guaranteed to launch from a clean slate. For example, by using a separate ephemeral web browser container for accessing a banking site, Apiary ensures that an already exploited web browser installation cannot compromise user privacy.

Ephemeral containers allow applications to launch other applications safely. For example, users often receive viewable email attachments such as PDF documents. To avoid compromising an email container, Apiary creates an ephemeral PDF viewer container for the PDF. Even if it is malicious, it cannot effect the user's desktop, as it only affects the ephemeral container.

Persistent containers are necessary for applications that maintain state across executions to prevent a single application compromise from affecting the entire system. Unlike ephemeral containers, users use only one persistent container per application. Some applications only use one type of container, while others use both. For example, email is typically used in a persistent container to maintain email state across executions. On the other hand, a web browser may be used both in a persistent container, to access a user's trusted websites, and in an ephemeral container, to view untrusted websites.

Apiary's containers work together to provide a security system that differs fundamentally from common security schemes that attempt to lock down applications within a restricted-privilege environment. In Apiary, each application container is an independent entity that is entirely isolated from every other application container on the Apiary desktop. One does not have to apply any security analysis or complex isolation rules to determine which files a specific application should be able to access. Also, in most other schemes, an application, once exploited, will continue to be exploited, even if the exploited application is restricted from accessing other applications' data. Apiary's ephemeral containers, however, prevent an exploit from persisting between application execution instances.

Apiary provides every desktop with two ways to share files between containers. First, containers can use standard file system share concepts to create directories that can be seen by multiple containers. This has the benefit of any data stored in the shared directory being automatically available to the other containers that have access to the share. Second, Apiary supplies every desktop with a special persistent container with a file explorer. The explorer has access to all of the user's containers and can manage all of the user's files, including copying them between containers. This is useful if the user wants to preserve a file from an ephemeral container, or move a file from one persistent container to another, as, for instance, when emailing a set of files. The file explorer container cannot be used in an ephemeral manner. Its functionality cannot be invoked by any other application on the system and no other application is allowed to execute within it. This prevents an exploited container from using the file explorer container to corrupt others.

It should be noted that both of these mechanism break, to some degree, the container's isolation. File system shares can be used by an exploited container as a vector to infect other containers by tricking a user into moving a malicious file between containers. However, this is a tension that will always exist in security systems that are meant to be usable to a diverse crowd of users. To mitigate this, Apiary lets documents stored in a persistent manner be viewable, by default, in an ephemeral container. For example, PDF files can be stored persistently, but always viewed in ephemeral containers. However, to persistently edit a PDF file, it would still have to be opened within a persistent container such that a malicious PDF would have a persistent effect.

3 Architecture

To support its container model, Apiary must have four capabilities. First, Apiary must be able to run applications within secure containers to provide application isolation. Second, Apiary must provide a single integrated display view of contains all running applications. Third, Apiary must be able to instantiate individual containers quickly and efficiently. Third, for a cohesive desktop experience, Apiary must allow applications in different containers to interact in a controlled manner.

Apiary does this by using a virtualization architecture that consists of three main components: an operating system container that provides a virtual execution environment, a virtual display system that provides a virtual display server and viewer, and the VLFS. Apiary also provides a desktop daemon that runs on the host to instantiate containers, manage their lifetimes, and ensure that they are correctly integrated.

3.1 Process Container

Apiary's containers enable applications to be isolated from one another. Individual applications can run in parallel within separate containers, and have no conception that there are other applications running. This enforces fault containment, as an exploited process only has access to files available within its own container.

Apiary's containers leverage commodity operating system features such as Solaris's zones [32], FreeBSD's jails [21], and Linux's containers [24] to create isolated and independent execution environments. Each container has its own private kernel namespace, file system, and display server, providing total isolation at the process, file system, and display levels. Programs within separate containers can only interact using normal network communication mechanisms. Each container is provided with an application control daemon that enables the virtual display viewer to query the container for its contents and interact with it.

3.2 Display

Apiary's virtual display system is crucial to complete process isolation and a cohesive desktop experience. In Apiary, each container has a virtual display similar to existing systems [4, 11, 37, 38]. This virtual display operates by decoupling the display state from the underlying hardware and enabling the display output to be redirected anywhere. This is necessary, since if containers were to share a single display directly, malicious applications could leverage built-in mechanisms in commodity display architectures [13, 27] to insert events and messages into other applications that share the display, enabling the malicious application to remotely control the others, effectively exploiting them as well. Many existing commodity security systems do not isolate applications at the display level, providing an easy avenue for attackers to further exploit applications on the desktop.

However, if each container's display is independent, they will not provide a single cohesive display. Apiary provides a cohesive display in two ways. First, it integrates the displays views into a single view. While a regular remote framework provides all the information needed to display each desktop, it assumes that there is no other display in use, and therefore expects to be able to draw the entire display area. In Apiary, where multiple containers are in use, this assumption does not hold. Therefore, to enable multiple displays to be integrated into a single view, the Apiary viewer does Porter-Duff compositing [30] of the displays using the *over* compositing operation.

Second, Apiary's display viewer provides the normal desktop metaphors that users expect, including a single menu structure for launching applications and an integrated task switcher that allows the user to switch among all running applications. Apiary leverages the application control daemon running within each container to enumerate all the available applications within the container, much like a regular menu application does in a traditional desktop. Instead of providing the menu directly in the screen, however, it transmits the collected data back to the viewer, which then integrates this information into its own menu, associating the menu entry with the container it came from. When a user selects a program from the viewer's menu, the viewer instructs the correct daemon to execute it within its container.

Similarly, to manage running applications effectively, Apiary provides a single taskbar with which the user can switch between all applications running within the integrated desktop. Apiary leverages the system's ability to enumerate windows and switch applications [15] to have the daemon enumerate all the windows provided by its container and transmit this information to the viewer. The viewer then integrates this information into a single taskbar with buttons corresponding to application windows. When the user switches windows using the taskbar, the viewer communicates with the daemon and instructs it to bring the correct window to the foreground.

3.3 Virtual Layered File System

Apiary requires containers to have file systems that are efficient in storage space, instantiating time, and management costs. A container's file system has to be efficient in storage space to enable regular desktops to support the large number of application containers that will be used within the Apiary desktop. A container's file system has to be efficient to instantiate to provide fast interactive response time, especially for launching ephemeral containers. Finally, a container's file system has to be efficient to manage as Apiary increases the number of file systems that are in use.

There are many existing file system approaches that could be used for Apiary, but they all suffer drawbacks. Package management [12, 35] is useful for managing a file system, however, it does not help provision a file system quickly nor is it space efficient if each independent container's file system has its own copy of the package. This also impacts management as each file system would have to be updated independently. File systems that support a branching semantic [7, 29] can be used to instantiate an ephemeral container quickly from a template file system. However, each template is independent and is therefore inefficient in space and in its ability to be maintained. Finally, even a single template file system with all the programs desired for every container does not help since it reduces isolation between programs.

Apiary introduces the concept of a VLFS to meet these requirements. The VLFS extends the package management concept to enable file systems to be created by composing shareable layers together into a single file system namespace view. VLFSs are built by combining a set of shared software layers together in a read-only manner with a per container private read-write layer. The VLFS's software layers are analogous to packages in a traditional system, and just like a file system will have hundreds of packages installed into it, a VLFS can be composed of hundreds of layers as well. Similar to a regular file system, where package management tools are used to update and install packages and their dependencies into the system, in Apiary, the same type of tools are used to create VLFSs and keep them up to date.

Unlike multiple regular file systems that will each need their own copy of a file, multiple VLFSs providing multiple applications are as efficient as a single regular file system as all files that are common among them will be stored once in the set of shared layers. Therefore, Apiary is able to store the file systems needed by its containers in an efficient manner. This also enables Apiary to manage its containers easily, as all one has to do is replace the single layer that contains the files that have to be updated to update each VLFS that uses it. The VLFS also enables Apiary to efficiently instantiate each container's file system. As no data has to be copied into place and each of the software layers is shared in a read-only manner, instantiating a file system is nearly instantaneous, and occurs transparently to the end user.

Layers are the primary building block of a VLFS. Layers are composed of three elements: the metadata files that describe the layers, configuration scripts that enable the layer to be added and removed from the VLFS correctly, and the primary component, its file system namespace. The layer's file system namespace is a self-contained set of files providing a specific set of functionality. The files are the individual items in the layer that are composed into a larger VLFS. There are no restrictions on the type of files. They can be regular files, symbolic links, hard links or device nodes. The layer's file system namespace can be viewed as a directory stored on the shared file system that contains the same file and directory structure that would be created if the individual items were installed into a traditional file system. On a traditional UNIX system, the directory structure would typically contain directories such as `/usr`, `/bin` and `/etc`. Symbolic links work as expected between layers since they work on path names, but a limitation is that hard links cannot exist between layers.

To support the VLFS, Apiary must solve a number of file system related problems. First, to enable quick instantiation, the VLFS must be able to quickly compose numerous distinct file system layers into a single static view. Second, as users expect to be able to interact with the VLFS as a normal file system, such as by creating and modifying files, Apiary has to enable an instantiated VLFS to be fully modifiable, while enforcing the read-only semantics for the software layers. Finally, Apiary has to support the ability to dynamically add and remove layers without taking the file system off-line. This is equivalent to installing, removing or upgrading a software package while a monolithic file system is online.

To solve these problems, Apiary leverages and expands upon unioning file systems [41]. Unioning file systems enable Apiary to solve the first problem as they allow the system to join multiple distinct file system namespaces into a single namespace view. These directories are unioned by layering directories on top of one another, joining all the files provide by all the layers into a single file system namespace view. As unioning requires no copying, it occurs quickly, enabling Apiary to be efficient in terms of provisioning.

To solve the second problem, union semantics are extended [41] to enable the assignment of properties to the layers, defining some layers to be read only, while others are read-write. This results in a model that borrows from copy-on-write (COW) file systems, where modifying a file on a lower read-only layer will cause it to be copied to the topmost writable layer in a COW fashion. The VLFS leverages this property to enable multiple VLFSs to share a set of software layers in a read-only manner, while providing each instantiated VLFS with its own read-write private layer to store file system modifications. This enables Apiary to be efficient in terms of storage.

This layering model also provides semantics that directory entries located at higher layers in the stack obscure the equivalent directory entries at lower levels. To provide a consistent semantic, if a file is deleted, a white-out mark is also created to ensure that files existing on a lower layer are not revealed. The white-out mechanism enables obscuring files on the read only lower layers, by just creating the white-out file on the topmost read-write private layer.

However, this creates a problem where a file deleted from a read-only share will never be able to be recreated. In a traditional file system, a deleted system file can be recovered by simply reinstalling the package that provided that file. In a VLFS, if the white-outs are stored in the private layer, they will persist, and even if the layer containing the file is replaced, the file will remain obscure. The VLFS solves this problem by associating individual private writable layers with each of its shared read-only layer for the storage of white-outs. When a file in a shared read-only layer is deleted, instead of writing a white-out file to the top-most layer, the white-out is stored in the shared layer's associated white-out layer. When a layer is replaced, its associated white-out layer will be replaced with an empty white-out layer as well, enabling any obscured file to be revealed.

Similarly, the VLFS has to handle the case where a file belonging to a shared read-only layer was modified and therefore copied up to the VLFS's private read-write layer. Apiary provides a *revert* command that enables the owner of a file that has been modified to revert the file's state to its original pristine state. While a regular VLFS *unlink* operation would remove the modified file from the private layer and create a white-out mark to obscure the original file, *revert* only removes the copy in the private layer thereby revealing the original copy below it.

Finally, VLFSs also have to support being managed while they are in use. In a traditional file system, an administrator can remove a package containing files in use, as deleting a file does not remove its contents from the file system until the file is no longer in use. However, if a layer is removed from a union, the data is effectively removed as well as unions only operate on system namespaces and not on the data the underlying files contain. If an administrator wanted to modify the VLFS by removing a layer due to deletion or upgrade maintenance, one would be forced to perform the maintenance off-line due to not being able to remove layers that are in use.

The VLFS solves this problem by emulating what the `unlink` operation does on a single file and applies it to layer removal. `unlink` operates in two steps. It first deletes the file name from the file system's namespace, while only freeing up the space taken up by the file's contents when it's no longer in use. Traditional package management systems rely on these semantics to enable them to upgrade packages, even if files are in use, by unlinking and then recreating them instead of directly overwriting the files. Apiary applies these semantics to layers. When a layer is removed from a VLFS, Apiary marks the layer as `unlinked`, removing it from the file system namespace. While this layer is no longer part of the file system namespace and therefore cannot be used by any operations that work on the file system namespace, such as `open`, it remains part of the VLFS enabling data operations, such as `read` and `write`, to continue to work correctly for files that were previously opened.

3.4 Inter-Application Integration

Apiary's isolated containers provide effective fault containment. However, isolated containers can hinder effective use of the desktop. For instance, if one's web browser is totally isolated from the PDF viewer, how does one view a downloaded PDF file? If the PDF viewer is included within the web browser container the isolation that should exist between web browser and an application viewing untrusted content is violated. Users could copy the file from the web browser container to the PDF viewer container, but this is not the integrated feel that users expect.

Apiary solves this problem by enabling applications in one container to execute specific applications in ephemeral containers. Every container is preconfigured with a list of programs that it enables other applications to use in an ephemeral manner. Apiary refers to these as *global* programs. For instance, a Firefox container can specify `/usr/bin/firefox` and a Xpdf container can specify `/usr/bin/xpdf` as global programs. Program paths marked global exist in all containers. Apiary accomplishes this by populating a single global layer, shared by all the container's VLFSs, with a wrapper program for each global program. This wrapper program is used to instantiate a new ephemeral container and execute the requested process within it.

When executed, the wrapper program determines how it was executed and what options were passed to it. It connects via network mechanisms to the Apiary desktop daemon on the same host and passes this information to it. The daemon maintains a mapping of global programs to containers and determines which container is being requested to be instantiated ephemerally. This ensures that only the specified global programs' containers will be instantiated, preventing an attacker from instantiating and executing arbitrary programs. Apiary is then able to instantiate the correct fresh ephemeral container, along with all the required desktop services, including a display server. The display server is then automatically connected to the viewer. Finally, the daemon executes the program as it was initially called in the new container.

To ensure that ephemeral containers are discarded when no longer needed, Apiary's monitors the process executed within the container. When it terminates, Apiary terminates the container. Similarly, as the Apiary viewer knows which containers are providing windows to it, if it determines that no more windows are being provided by the container, it instructs the desktop daemon to terminate the container. This ensures that an exploited process does not continue running in the background.

However, running a new program in a fresh container is not enough to integrate applications correctly. When Firefox downloads a PDF and executes a PDF viewer, it must enable the viewer to view the file. This will fail because Firefox and an ephemeral PDF viewer containers do not share the same file system. To support this functionality, Apiary enables small private read-only file shares between a parent container and the child ephemeral container it instantiates. Because well-behaved applications such as Firefox, Thunderbird, and OpenOffice only use the system's temporary directory to pass files among them, Apiary restricts this automatic file sharing ability to files located under `/tmp`. To ensure that there are no namespace conflicts between containers, Apiary provides containers with their own private directory under `/tmp` to use for temporary files, and they are preconfigured to use that directory as their temp directory.

But providing a fully shared temporary file directory allows an exploited container to access private files that are placed there when passed to an ephemeral container. For instance, if a user downloads a malicious PDF and a bank statement in close succession, they will both exist in the temp directory at the same time. To prevent this, Apiary provides a special file system that enhances the read-only shares with an access control list (ACL) that determines which containers can access which files. By default, these directories will appear empty to the rest of the containers, as they do not have access to any of the files. This prevents an exploited container from accessing data not explicitly given to it. A file will only be visible within the directories if the Apiary desktop daemon instructs the file system to reveal that file by adding the container to the file's ACL. This occurs when a global program's wrapper is executed and the daemon determines that a file was passed to it as an option. The daemon then adds the ephemeral container to the file's ACL. Because the directory structure is consistent between containers, simply executing the requested program in the new ephemeral container with the same options is sufficient.

Situations can conceivably exist where the ephemeral application would need to access multiple files located within the temporary directory, such as a web page with images where the entire web page is saved. In these cases, Apiary's sharing will fail to permit access to all the files. However, in practice, these situations are uncommon and Apiary's scheme works well. In situations where this can occur, one can construct the application containers to contain all the programs needed. For instance, in a web development container, one will provide a web browser to preview one's content, instead of instantiating an external ephemeral container, thereby preventing this problem from occurring.

Apiary enables the file explorer container discussed in Section 2 in a similar way. The file explorer container is similar to Apiary's other containers. It is fully isolated from the rest of the containers and users interact with it via the regular display viewer. However, the other containers are not fully isolated from it. This is necessary as users can store their files in multiple locations in each container, most notably, the `/tmp` directory and the user's home directory. Apiary's file explorer provides read-write access to each of these areas as file shares within the file explorer's file system namespace. Apiary prevents any executable located within these file systems from executing with the file explorer container to prevent malicious programs from exploiting it. Users are able to use normal copy/paste semantics to move files among containers. While this is more involved than a normal desktop with only a single namespace, users generally do not have to move files among containers.

The primary situation in which users might desire to move files between containers is when interacting with an ephemeral container, as a user might want to preserve a file from there. For instance, users can run web browsers in an ephemeral containers to maintain privacy, but also download files they want to keep. While the ephemeral container is active, a user can just use the file explorer to view all active containers. To avoid situations where users only remembers after terminating the ephemeral container that it had files they wanted to keep, Apiary archives all newly created or modified non hidden files that are accessible to the file explorer when the ephemeral container terminates. This allows a user to gain access to them even after the ephemeral container has terminated. Apiary automatically trims this archive if no visible data was stored within the ephemeral container, such as in the case of an ephemeral web browser that the user only used to view a web page, not download and save a specific file. Similarly, Apiary provides users the ability to trim the archive to remove ephemeral container archives that do not contain data they need.

Apiary also turns the desktop viewer into an interprocess communication (IPC) proxy that can enable IPC state to be shared among containers in a controlled and secure manner. This means that only explicitly allowed IPC state is shared. For example, one of the most basic ways desktop applications share state is via the shared desktop clipboard. To handle the clipboard, each container's desktop daemon monitors the clipboard for changes. Whenever a change is made to one container's clipboard, this update is sent to the Apiary viewer, and then propagated to all the other containers. The Apiary viewer also keeps a copy of the clipboard so that any future container can be initialized with the current clipboard state. This enables users to continue to use the clipboard with applications in different containers in a manner that is consistent with a traditional desktop. However, by allowing the clipboard of an ephemeral container to read from the shared clipboard, Apiary does allow information to be leaked. This can be handled by only allowing ephemeral containers to write to the shared clipboard, if the decreased functionality is acceptable.

4 Experimental Results

We have implemented a remote desktop Apiary prototype system for the Linux desktop without any application, library, window system, or base kernel changes. The prototype consists of a virtual display driver for the X window system based on MetaVNC [37], a set of user space utilities that enable container integration, and a loadable kernel module for the Linux 2.6 kernel that provides the ability to create and mount VLFSs. Apiary uses Zap [28], a predecessor to Linux containers [24], to provide the isolated containers.

For our prototype, we created 211 software layers by converting the set of Debian packages needed by the set of applications we tested into individual layers. Each Debian package can be viewed as providing three sets of items, a set of files that is extracted into a file system, a set of metadata that determines the dependency relationship among packages, and configuration scripts that are executed on installation and removal to ensure the packages are installed correctly. For the VLFS, we first extract the set of files into a directory that will be used for composition. Second, we extract the metadata that determines dependency relationships between the packages and associate it with the newly created layers. Finally, we associate the configuration scripts from each package with the layers which are used each time the layer is added or removed from an application appliance. Using these layers, we are able to create per application appliances for each individual application by simply selecting which high-level applications we want within the appliance, such as Firefox, with the dependencies between the layers ensuring that all the required layers are included. Using these appliances, we are able to instantly provision persistent and ephemeral containers for the applications as needed.

Using this prototype, we used real exploits to evaluate Apiary's ability to contain and recover from attacks. We conducted a user study to evaluate Apiary's ease of use compared to a traditional desktop. We also measured Apiary's performance with real applications in terms of runtime overhead, startup time, and storage efficiency. For our experiments, we compared a plain Linux desktop with common applications installed to an Apiary desktop that has applications available for use in persistent and ephemeral containers. The applications we used are the Pidgin 2.4.3 instant messenger, the Firefox 3.0.3 web browser, the Thunderbird 2.0 email client, the OpenOffice.org 2.4.1 office suite, the Xpdf 3.02 PDF viewing program, and the MPlayer 1.0-rc2 media player. Experiments were conducted on an IBM HS20 eServer blade with dual 3.06 GHz Intel Xeon CPUs and 2.5 GB RAM. All desktop application execution occurred on the blade. Participants in the usage study connected to the blade via a Thinkpad T42p laptop, with a 1.8 GHz Intel Pentium-M CPU and 2GB of RAM running the MetaVNC viewer.

4.1 Handling Exploits

We tested two scenarios that illustrate Apiary's ability to contain and recover from a desktop application exploit, as well as explore how different decisions can affect the security of Apiary's containers.

4.1.1 Malicious Files

Many desktop applications have been shown to be vulnerable to maliciously created files that enable an attacker to subvert the target machine and destroy data. These attacks are prevalent on the Internet, as many users will download and view whatever files are sent to them. To demonstrate this problem, we use 2 malicious files [14, 16] that exploit Xpdf 1.01 and mpg123 pre0.59s. We installed the older Xpdf version in the Xpdf container and mpg123 in the MPlayer container. The mpg123 exploit works by creating an invalid mp3 file that triggers a buffer overflow in old versions of mpg123, enabling the exploit to execute any program it desires. The Xpdf exploit works by exploiting a behavior of how Xpdf launched helper programs, that is, by passing a string to `sh -c`. By including a back-tick (` `) string within a URL embedded in the PDF file, an attacker could get Xpdf to launch unknown programs. Both of these exploits are able to leverage sudo to perform privileged tasks, in this case, deleting the entire file system. Sudo is exploited because popular distributions require users to use it to gain root privileges and have it configured to run any applications. Additionally, sudo, by default, caches the user's credentials to avoid needing to authenticate the user each time it needs to perform a privileged action. However, this enables local exploits to leverage the cached credentials to gain root privileges.

In the plain Linux system, recovering from these exploits required us to spend a significant amount of time reinstalling the system from scratch, as we had to install many individual programs, not just the one that was exploited. Additionally, we had to recover a user's 23 GB home directory from backup. Reinstalling a basic Debian installation took 19 minutes. However, reinstalling the complete desktop environment took a total of 50 minutes. Recovering the user's home directory, which included multimedia files, research papers, email, and many other assorted files, took an additional 88 minutes when transferred over a Gbps LAN.

Apiary protected the desktop and enabled easier recovery. It protected the desktop by letting the malicious files be viewed within ephemeral containers. Even though the exploits proceeded as expected and deleted the container's entire file system, the damage caused is invisible to the user, because that ephemeral container was never used again. Even when we permitted the exploits to execute within persistent containers, Apiary enabled significantly easier recovery from the exploits. As shown in Table 2, Apiary can provision a file system in just a few milliseconds. This is nearly 6 orders of magnitude faster than the traditional method of recovering a system by reinstallation. Furthermore, Apiary's persistent containers divide up home directory content between them. For instance, a web browser container's home directory will contain the web browser's configuration, browser cache and downloaded files, while a word processing container will contain the documents one has created or edited. This eliminates the need to recover all of a user's data if only one application is exploited.

This also shows how persistent containers can be constructed in a more secure manner to prevent exploits from harming the user. As a large amount of the above user's data, such as media files, is only accessed in a read-only manner, the data can be stored on file system shares. This enables the user to allow the different containers to have different levels of access to the share. The file explorer container can access it in a read-write manner, enabling a user to manage the contents of the file system share, while the actual applications that view these files can be restricted to accessing them in a read-only manner, protecting the files from being damaged by exploits.

4.1.2 Malicious Plugins

Applications are also exploited via malware that users are tricked into downloading and installing. This can be an independent program or a plugin that integrates with an already-installed application. For example, malware have tried to convince users to download a "codec" they need to view a video. Recently, a malicious Firefox extension was discovered [6] that leverages Firefox's extension and plugin mechanism to extract a user's banking credentials from the browser when the user visits a bank's website. These attacks are common because users are badly conditioned to always allow a browser to install what it claims is needed. When installed into a traditional environment, this malicious extension persists until the user, or the user's anti-virus software, discovers and removes it. As it does not affect regular use of the browser, there is little to alert users that they have been attacked. As this exploit is not publicly available, we simulated its presence with the non-malicious Greasemonkey Firefox extension [18]. Like with the malicious file, ephemeral containers prevented the extension from persisting.

However, this exploit poses a significant risk to persistent web browser containers. While one might expect Firefox extensions to be uninstallable through Firefox's extension manager, this is only true of extensions that are installed through it. If an extension is installed directly into the file system, it can only be disabled this way, but not uninstalled. This applies equally to Apiary and traditional machines. While Apiary users can quickly recreate the entire persistent Firefox container, that requires knowing that the installation was exploited. Apiary handles this situation more elegantly by allowing the user to use Firefox in multiple web browsing containers. In this case, we created a general-purpose web browsing container for regular use, as well as a financial web browsing container for the bank website only. Apiary configured the financial web browser container to refuse to install any addons within it's browser, keeping it isolated and secure even when the general-purpose web browsing container was compromised.

Apiary enables the creation of multiple independent application containers, each containing the same application, but performing different tasks, such as visiting a bank website. Because the great majority of the VLFS's layers are shared, the user incurs very little cost for these multiple independent containers. This approach can be extended to other related but independent tasks, for instance, using a media player to listen to one's personal collection of music, as opposed to listening to Internet radio from an untrusted source.

This scenario also reveals a problem with how plugins and other extensions are currently handled. When the browser provides its own package management interface independent of the system's built-in package manager, this impacts Apiary, because certain application extensions might be needed in an ephemeral container, but if they are not known to the package manager, they cannot be easily included. However, many plugins and browser extensions are globally installable and manageable via the package manager itself in systems like Debian. In these systems, this yields the benefit that when multiple users wish to use an extension, it only has to be installed once. In Apiary, it additionally provides the benefit that it can become part of the application container's definition, making it available to the ephemeral container without requiring it to be manually installed by the user on each ephemeral execution.

4.2 Usage Study

We performed a 24-person usage study that evaluated the ability of users to use Apiary's containerized application model based on our prototype environment, focusing on their ability to execute applications from within other programs. Participants were mostly recruited from within our local university, including faculty, staff and students. All of the users were computer-literate, though a significant number were not power users and included business and humanities students.

For our study, we created three distinct environments. The first was a plain Linux environment running the Xfce4 desktop. It provided a normal desktop Linux experience with a background of icons for files and programs and a full-fledged panel application with a menu, task switcher, clock and other assorted applets. The second was a full Apiary environment. It provided a much sparser experience, as the current Apiary prototype only provides a set of applications and not a full desktop envi-

Test	Description
Untar	Extract Linux 2.6.19 kernel source archive
Gzip	Compress a 250 MB Linux kernel source archive
Octave	Octave 3.0.1 numerical benchmark [19]
Kernel	Build the 2.6.19 kernel

Table 1: Application Benchmarks

ronment. The third was a neutered Apiary environment that differs from the full environment in not launching any child applications within ephemeral containers.

The three environments enable us to compare the participants' experience along two axes. First, we can compare the plain Linux environment, where each application is only installed once and always run from the same environment, to the neutered Apiary environment, where each application is also only installed once and run from the same environment. This allows us to measure the cost of using the Apiary viewer, with its built-in taskbar and application menu, against plain Linux, where the taskbar and application menu are regular applications within the environment. Second, the full and neutered Apiary desktops enable us to isolate the actual and perceived cost to the participants of instantiating ephemeral containers for application execution. We presented the environments to the participants in random order and iterated through all 6 permutations equally.

We timed the participants as they performed a number of specific multi-step tasks in each environment that were designed to measure the overhead of using multiple interacting applications. In summary, the tasks were: (1) download and view a PDF file with Firefox and Xpdf and follow a link embedded in the PDF back to the web; (2) read an email in Thunderbird that contains an attachment that is to be edited in OpenOffice and returned to the sender; (3) create a document in OpenOffice that contains text copied and pasted from the web and sent by e-mail as a PDF file; (4) create a "Hello World" web page in OpenOffice and preview it in Firefox; and (5) launch a link received in the Pidgin IM client in Firefox.

As Figure 2 shows, the average time to complete each task only differed by a few seconds for all tasks in all environments. Figure 2 shows that even in tasks where users were creating multiple new ephemeral containers, that overhead imposed in creating these containers is minimal and generally unnoticeable to the user. Therefore, users were able to complete the tasks using Apiary with the same efficiency as on a regular system.

The participants also rated their perceived ease of use of each environment on a scale of 1 to 5. The average rating, of both the plain Linux environment and Apiary, was a 3.9 with a standard deviation of 0.9 and 1.1 respectively. The participants were asked if they could imagine using Apiary full time and whether they would prefer to do so if it would keep their desktop more secure. All of the participants expressed a willingness to use this environment full-time, and a large majority indicated that they would prefer to use Apiary over the plain Linux environment if it would keep their data more secure.

4.3 Performance Measurements

4.3.1 Application Performance

To measure the performance overhead of Apiary on real applications, we compared the runtime performance of a number of applications within the Apiary environment against their performance in a traditional environment. To provide a conservative measure of Apiary performance, we used a container with all of the applications from all of our experiments to maximize the number of layers installed.

Table 1 lists our application tests. We focus mostly on file system benchmarks, as we have shown [4, 28] that display and operating system virtualization have little overhead. Untar tests file creation and throughput. Gzip tests file system throughput and computation. Octave is a pure computation benchmark. The kernel build tests computation and stresses the file system, because of the large number of lookups that occur due to the large size of the kernel source tree and the repeated execution of the preprocessor, compiler, and linker. To stress the system with many containers and provide a conservative performance measure, each test was run in parallel with 25 instances. To avoid out-of-memory (OOM) conditions, as the Octave benchmark requires 100 to 200 MB of memory at various points during its execution, we ran the benchmarks staggered 5 seconds apart to ensure they kept their high memory usage areas isolated to avoid the benchmarks being killed by Linux's OOM handler. Figure 3 shows that Apiary imposes almost no overhead in most cases, with about 10% overhead in the kernel build case due to the VLFS's constant need to perform lookups on the file system incurring an extra cost. This demonstrates that Apiary is able to scale to a large number of concurrent containers with minimal overhead.

4.3.2 Container Creation

For ephemeral containers to be useful, container instantiation must be quick and impose little overhead on application startup time. Although our user study already indicates that Apiary container instantiation overhead is not noticeable to users, we measure the overhead in two more ways. We measure both how long it takes to instantiate a VLFS and how long the application takes to start up within the container. First, we compare how long it takes to setup a VLFS against three other potential approaches to setting up the same container file system: using traditional Debian bootstrapping tools (**Create**), extracting the file system from a tar archive (**Extract**), and using Btrfs [9], a file system with a snapshot operation, to create a new snapshot and branch of a preexisting file system namespace (**FS-Snap**). To minimize network effects with the bootstrapping tools, we used a local Debian mirror on the local 100 Mbps campus network, and were able to saturate the connection while fetching the packages to be installed.

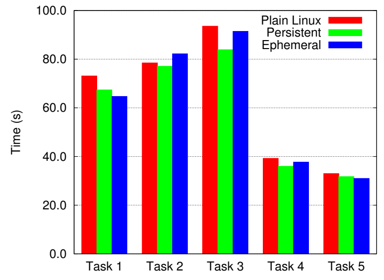


Figure 2: Usage Study Task Times

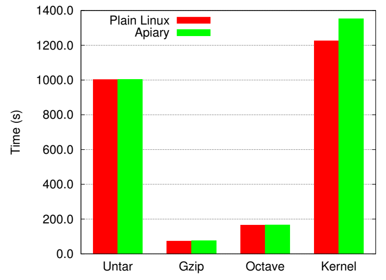


Figure 3: Overhead at Scale

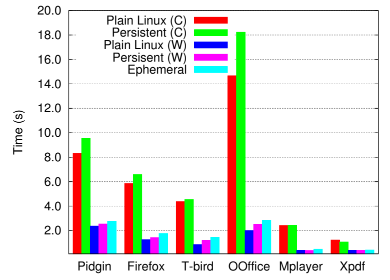


Figure 4: Application Startup Time

	P	F	T	O	X	M
Create (s)	317	276	294	365	291	294
Extract (s)	82	86	87	150	81	81
FS-Snap (s)	.016	.015	.016	.020	.009	.010
Apiary (s)	.005	.005	.005	.005	.005	.005

Table 2: File System Instantiating Times for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

Table 2 shows that Apiary instantiates containers with a VLFS composed of nearly 200 layers nearly instantaneously. This compares very positively with traditional ways of setting up a system. Table 2 show that it takes a significant amount of time to create a file system for the application container using either Debian's bootstrapping tool or extracting it from a tar archive. Therefore, these methods are not usable for ephemeral application containers, as users will not want to wait minutes for their applications to start. Tar archives also suffer from their need be actively maintained and rebuilt whenever they need fixes. Therefore, the amount of administrative work increases linearly with the number of applications in use. As Apiary creates the file system nearly instantaneously, it is able to support the creation of ephemeral application containers with no noticeable overhead to the users. While Table 2 shows that file systems with a snapshot and branch operation can also perform quickly, the user would have to manage each of the application's independent file systems separately.

Second we quantify startup time by measuring how long it takes for the application to open and then be automatically closed using ephemeral containers, persistent containers, and plain Linux. In the case of Firefox, Xpdf, and OpenOffice.org, this includes the time it takes to display the initial page of a document, while Pidgin, MPlayer and Thunderbird are only loading the program. For ephemeral containers, we measure the total time it takes to set up the container and execute the application within it, while for persistent containers and plain Linux, we only measure application execution time as these environments are persistent and therefore require no setup time. We compare ephemeral container application startup time to cold (C) and warm (W) cache application startup times for both plain Linux and Apiary's persistent containers. We include cold cache results for benchmarking purposes and warm cache results to demonstrate the results users would normally see.

As Figure 4, shows, the startup time overhead of running within a container versus plain Linux with no containers is generally under 25% in both cold and warm cache scenarios. This overhead is mostly due to the added overhead of opening the many files needed by today's complex applications. The most complex application, OpenOffice, imposes the most, while the least complex application, Xpdf, has negligible overhead. While the maximum absolute extra time spent in the cold cache case was nearly 5 seconds for OpenOffice, in the warm cache case it dropped to under .5 seconds. Ephemeral containers provide an interesting result. Even though they have a fresh new file system and would be thought to be equivalent to a cold cache startup, they are nearly equivalent to the warm cache case. This is because their underlying layers are already cached by the system. The ephemeral case has a slightly higher overhead due to the need to create the container and execute a display server inside of it in addition to regular application startup. However, as this takes under 10 milliseconds, it adds only a minimal amount to the ephemeral application startup time.

4.4 File System Efficiency

To support a large number of containers, Apiary must store and manage its file system efficiently. This means that storage space should not significantly increase with an increasing number of instantiated containers and should be easily manageable in terms of application updates. For each application's VLFS, Table 3 shows its size, its number of layers, the amount of state shared with the other application VLFSs, and the amount of state unique to it. For instance, the 129 layers that make up Firefox's VLFS require 353 MB, of which 330 MB are

	P	F	T	O	X	M
Size (MB)	394	353	367	645	339	355
# Layers	147	129	125	186	130	162
Shared (MB)	322	330	335	329	330	326
Unique (MB)	72	23	32	316	9	29

Table 3: VLFS Layer Storage Breakdown for (P)idgin, (F)irefox, (T)hunderbird, (O)penOffice, (X)pdf and (M)Player

shared with other applications and 23 MB are unique to the Firefox VLFS. Table 3 shows that the majority of files in each container are shared with other containers.

Table 4 compares the storage requirements of a plain Linux desktop versus Apiary when using different numbers of containers to store the six applications listed in Table 3. When all the applications are installed within a single container, plain Linux and Apiary require the same amount of storage. However, when each application is installed within its own container, Apiary's VLFS imposes no additional storage requirements while the traditional Linux method of provisioning an independent file system for each container requires more than three times more disk space due to the duplication of files amongst the containers. If instead of using local desktops, multiple remote desktops are provided on a server, the VLFS usage would remain constant with the total size of all layers, while the plain Linux case would grow linearly with the number of desktops.

Table 4 demonstrates how Apiary improves the ability of users to maintain their many containers. We measured the time it took to apply a security update common to all the containers. Table 4 shows the time it took to update a single container containing all the applications, as well as all six application containers. The plain Linux case is two order of magnitude longer due its need to extract files from a package archive and copy them into the container's file system. In Apiary, no copying has to be performed. While Table 4 demonstrates that an individual update by itself does not take too long, the total time to apply common updates to many containers rises linearly with the number of containers.

5 Related Work

Isolation mechanisms such as VMs [39] and operating system containers [24, 32] have long been used to increase the security of applications. However, this results in applications not being integrated into the user's desktop experience. Each application is totally independent and cannot leverage another one. Products like VMware's Unity [39] attempt to solve part of this issue by combining the applications from multiple VMs into a single display with a single menu and taskbar, as well as providing file system sharing between host and VMs. While VMs provide superior isolation, they suffer higher

		FS Size	Update Time
1 Container	**Plain Linux**	815 MB	18 s
	Apiary	815 MB	124 ms
6 Containers	**Plain Linux**	2453 MB	108 s
	Apiary	815 MB	745 ms

Table 4: Apiary vs Traditional File System Efficiency

overhead due to running independent operating systems. This impacts performance and makes them unusable for ephemeral usage on account of their long startup times. In contrast, Apiary provides lightweight containers that can support ephemeral execution.

Tahoma [34] is similar to Apiary in that it creates fully isolated application environments that remain part of a single desktop. Tahoma creates browser applications that are limited to certain resources, such as specific URLs, and that are fully isolated from each other. However, it only provides these isolated application environments for web browsers. It does not provide any way to integrate these isolated environments and does not provide ephemeral application environments. Google's Chrome web browser [17] builds upon some of these ideas to isolate web browser pages within a single browser. But the browser as a whole does not offer any isolation from the system. While its multiple-process model uses operating system mechanisms to isolate separate web pages that are concurrently viewed, it does not provide any isolation from the system itself. For instance, any plugin that is executed has the same access to the underlying system as does the user running the browser.

Modern web browsers improve privacy by providing private browsing modes that prevent browser state from being committed to disk. While they serve a similar purpose to ephemeral containers, private browsing is fundamentally different. First, it has to be written into the program itself. Many different types of programs have privacy modes to prevent them from recording state and this model requires them to implement it independently. Second, it only provides a basic level of privacy. It cannot prevent a plugin from writing state to disk. Furthermore, it makes the entire browser and any helper program or plugin that it executes part of the trusted computing base (TCB). This means that the user's entire desktop becomes part of the TCB. If any of those elements gets exploited, no privacy guarantees can be enforced. Apiary's ephemeral containers make the entire execution private and support any application with a state a user desires to remain private without any application modifications. It also keeps the TCB much smaller, by only requiring that the underlying operating system kernel and the minimal environment of Apiary's system daemon be trusted.

Apiary's ability to run multiple applications in parallel resembles Lampson's Red/Green isolation [22] and WinDowBox [3]. These schemes involve users running two

or more separate environments, for instance, a red environment for regular usage and a green environment for actions requiring a higher level of trust. However, unlike Apiary's ephemeral containers, if an exploit enters the green container, it will persist. Furthermore, by requiring two separate virtual machines, one increases the amount of work a user has to do to manage their machines. Apiary, by leveraging the VLFS, minimizes the overhead required required to manage multiple machines. Storage Capsules [8] also attempts to mitigate this problem by securely running the applications requiring trust in the same operating system environment as the untrusted applications, while keeping their data isolated from one another. However, this involves significant startup and teardown costs for each execution.

File systems and block devices with branching or COW semantics [7, 29, 36] can be used to create a fresh file system namespace for a new container quickly. However, these file systems do not help to manage the large number of containers that exist within Apiary. Because each container has a unique file system with different sets of applications, administrators must create individual file systems tailored to each application. They cannot create a single template file system with all applications because applications can have conflicting dependency requirements or desire to use the same file system path locations. Furthermore, if all applications are in a single file system, they are not isolated from each other. This results in a set of space-inefficient file systems, as each file system has an independent copy of many common files. This inefficiency also makes management harder. When security holes are discovered and fixed, each individual file system must be updated independently.

Many systems have been created that attempt to provide security through isolation mechanisms [1, 5, 10, 23, 25, 31, 33, 40]. All these systems differ from Apiary in that they try to isolate the many different components that make up a standard fully-integrated single system using sets of rules to determine which of the machine's resources the application should be able to access. This often results in one of two outcomes. First, a policy is created that is too strict and does not let the application run correctly. Second, a policy is created that is too lenient and lets an exploited application interact with data and applications it should not be able to access. Apiary, on the other hand, forces each components to be fully isolated within its own container before determining on which levels it should be integrated. As container setup leverages regular installation utilities to ensure all the required components are installed, it is much easier to ensure the container is setup correctly and provides all the resources that the application needs to execute. As the container is independent from all other containers on the system, no complicated rule sets have to be created to determine what it needs access to. Furthermore, rule based systems do not provide ephemeral execution and therefore if an application gets exploited, it will remain exploited, even if the exploit is confined.

Solitude [20] provides isolation via its Isolation File System (IFS), which a user can throw away. This is similar to Apiary's ephemeral containers. However, the IFSs are not fully isolated. First, Solitude does not create a new IFS for each application execution. Second, the IFS is built on top of a base file system with which it can share data, breaking the isolation. To handle this, Solitude implements taint tracking on files shared with the underlying base file system. This helps determine post facto what other applications may have been corrupted. Similarly, Solitude only provides isolation at the file system level. Because each application still shares a single display, malicious and exploited applications can leverage built-in mechanisms in commodity display architectures [13, 27] to insert events and messages into other applications sharing the display.

6 Conclusions

Apiary introduces a new compartmentalized application desktop paradigm. Instead of running one's applications in a single environment with complex rules to isolate the applications from each other, Apiary enables them to be easily and completely isolated while retaining the integrated feel users expect from their desktop computer. The key innovations that make this possible are the introduction of the Virtual Layered File System and the ephemeral containers they enable. The Virtual Layered File System enables the multiple containers to be stored as efficiently as a single regular desktop, while also allowing containers to be instantiated almost instantly. This functionality enables the creation of the ephemeral containers that provide an always fresh and clean environment for applications to run in. Ephemeral containers prevent malicious data from having any persistent effect on the system and isolate faults to a single application instance.

We have implemented Apiary on Linux without requiring any operating system kernel or application changes. Our results demonstrate that Apiary's containerized desktop severely reduces the threat posed by malicious files and plugins by isolating them in ephemeral containers and enabling users to quickly recover if they penetrate a persistent container. Our 24-person usage study demonstrates that Apiary is as easy to use as a regular Linux desktop by both measuring the time it took users to perform their tasks and their subjective opinions. Furthermore, we demonstrate that Apiary adds minimal overhead to application performance, is as efficient as a regular desktop in its use of storage space, and instantiates ephemeral containers in less than .5 s.

7 Acknowledgments

This work was supported in part by NSF grants CNS-0717544, CNS-0914845 and CNS-0905246, and AFOSR MURI grant FA9550-07-1-0527.

References

[1] A. Acharya and M. Raje. MAPbox: Using Parameterized Behavior Classes to Confine Applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000.

[2] Adobe Systems Incorporated. Buffer Overflow Issue in Versions 9.0 and Earlier of Adobe Reader and Acrobat. http://www.adobe.com/support/security/advisories/apsa09-01.html, Feb. 2009.

[3] D. Balfanz and D. R. Simon. WindowBox: A Simple Security Model for the Connected Desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle, WA, Aug. 2000.

[4] R. A. Baratto, L. N. Kim, and J. Nieh. THINC: A Virtual Display Architecture for Thin-Client Computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, United Kingdom, Oct. 2005.

[5] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific File Protection for the UNIX Operating System. In *Proceedings of the 1995 USENIX Winter Technical Conference*, New Orleans, LA, Jan. 1995.

[6] bitdefender. Trojan.pws.chromeinject.b. http://www.bitdefender.com/VIRUS-1000451-en--Trojan.PWS.ChromeInject.B.html, Nov. 2008.

[7] J. Bonwick and B. Moore. ZFS: The Last Word In File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, Nov. 2005.

[8] K. Borders, E. V. Weele, B. Lau, and A. Prakash. Protecting Confidential Data on Personal Computers with Storage Capsules. In *Proceedings of the 18th USENIX Security Symposium*, Montreal. Canada, Aug. 2009.

[9] Btrfs. https://btrfs.wiki.kernel.org.

[10] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, LA, Dec. 2000.

[11] B. Cumberland, G. Carius, and A. Muir. *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*. Microsoft Press, Aug. 1999.

[12] J. Fernandez-Sanguino. Debian GNU/Linux FAQ - Chapter 8 - The Debian Package Management Tools. http://www.debian.org/doc/FAQ/ch-pkgtools.en.html.

[13] J. Gettys and R. W. Scheifler. *Xlib - C Language X Interface*. X Consortium, Inc., 1996.

[14] M. Gilmore. 10Day CERT Advisory on PDF Files. http://seclists.org/fulldisclosure/2003/Jun/0463.html, June 2003.

[15] Gnome.org. Libwnck Reference Manual. http://library.gnome.org/devel/libwnck/.

[16] GOBBLES Security. Local/Remote Mpg123 Exploit. http://www.opennet.ru/base/exploits/1042565884_668.txt.html, Jan. 2003.

[17] Google. Google Chrome - Features. http://www.google.com/chrome/intl/en/features.html.

[18] Greasemonkey. http://www.greasespot.net/.

[19] P. Grosjean. Speed Comparison of Various Number Crunching Packages (Version 2). http://www.sciviews.org/benchmark/, Mar. 2003.

[20] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level Isolation and Recovery with Solitude. In *Proceedings of the 3rd ACM European Conference on Computer Systems*, Glasgow, Scotland, Apr. 2008.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proceedings of the 2nd International SANE Conference*, MECC, Maastricht, The Netherlands, May 2000.

[22] B. Lampson. Accountability and Freedom. http://research.microsoft.com/en-us/um/people/blampson/slides/accountabilityandfreedom.ppt, Sept. 2005.

[23] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, Dec. 2003.

[24] Linux Containers. http://lxc.sourceforge.net/.

[25] P. Loscocco and S. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 2001 USENIX Annual Technical Conference: FREENIX Track*, Boston, MA, June 2001.

[26] Microsoft Application Virtualization. http://www.microsoft.com/systemcenter/appv/.

[27] Microsoft Corp. SendMessage Function. http://msdn.microsoft.com/en-us/library/ms644950.aspx.

[28] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[29] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization Aware File Systems: Getting Beyond the Limitations of Virtual Disks. In *Proceedings of the 3rd Symposium of Networked Systems Design and Implementation*, San Jose, CA, May 2006.

[30] T. Porter and T. Duff. Compositing Digital Images. *Computer Graphics*, 18(3):253–259, July 1984.

[31] S. Potter, J. Nieh, and M. Selsky. Secure Isolation of Untrusted Legacy Applications. In *Proceedings of the 21st Conference on Large Installation System Administration*, Dallas, TX, Nov. 2007.

[32] D. Price and A. Tucker. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th Large Installation System Administration Conference*, Atlanta, GA, Nov. 2004.

[33] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, Aug. 2003.

[34] C. Reis and S. D. Gribble. Isolating Web Programs in Modern Browser Architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, Nuremberg, Germany, Mar. 2009.

[35] RPM Package Manager. http://www.rpm.org/.

[36] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, Boston, MA, Dec. 2002.

[37] S. Uchino. MetaVNC - A Window Aware VNC. http://metavnc.sourceforge.net/.

[38] Virtual Network Computing. http://www.realvnc.com/.

[39] VMware Inc. VMware Worksation 6.5 Release Notes. http://www.vmware.com/support/ws65/doc/releasenotes_ws65.html, Oct. 2008.

[40] D. Wagner. Janus: An Approach for Confinement of Untrusted Applications. Master's thesis, University of California, Berkeley, Aug. 1999.

[41] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in Namespace Unification. *ACM Transactions on Storage*, 2(1):1–32, Feb. 2006.

Tolerating Malicious Device Drivers in Linux

Silas Boyd-Wickizer and Nickolai Zeldovich
MIT CSAIL

Abstract

This paper presents SUD, a system for running existing Linux device drivers as untrusted user-space processes. Even if the device driver is controlled by a malicious adversary, it cannot compromise the rest of the system. One significant challenge of fully isolating a driver is to confine the actions of its hardware device. SUD relies on IOMMU hardware, PCI express bridges, and message-signaled interrupts to confine hardware devices. SUD runs unmodified Linux device drivers, by emulating a Linux kernel environment in user-space. A prototype of SUD runs drivers for Gigabit Ethernet, 802.11 wireless, sound cards, USB host controllers, and USB devices, and it is easy to add a new device class. SUD achieves the same performance as an in-kernel driver on networking benchmarks, and can saturate a Gigabit Ethernet link. SUD incurs a CPU overhead comparable to existing runtime driver isolation techniques, while providing much stronger isolation guarantees for untrusted drivers. Finally, SUD requires minimal changes to the kernel—just two kernel modules comprising 4,000 lines of code—which may at last allow the adoption of these ideas in practice.

1 Introduction

Device drivers are a significant source of bugs in an operating system kernel [11, 13]. Drivers must implement complex features, such as wireless drivers running the 802.11 protocol, the Linux DRM graphics subsystem supporting OpenGL operations, or the Linux X server running with direct access to the underlying hardware. Drivers must correctly handle any error conditions that may arise at runtime [19]. Finally, drivers must execute in restrictive kernel environments, such as when interrupts are disabled, without relying on commonly-available services like memory allocation. The result is kernel crashes due to bugs in drivers, and even security vulnerabilities that can be exploited by attackers [1, 5].

Significant work has been done on trying to isolate device drivers, and to make operating systems reliable in the face of device driver failures [4, 6, 7, 10, 12, 14, 15, 21–23, 26–29, 33–36]. Many research operating systems include support to fully isolate device drivers [2, 15]. Unfortunately, work on commodity operating systems, like Linux, focuses on fault isolation to prevent common device driver bugs, and cannot protect the rest of the system from malicious device drivers [7, 30]. For instance, many driver isolation techniques *trust* the driver not to subvert the isolation, or not to livelock the system. If attackers exploit a bug in the device driver [1, 5], they can proceed to subvert the isolation mechanism and compromise the entire system. While some systems can provide stronger guarantees [28, 33], they rely on the availability of a fully-trusted, precise specification of the hardware device's behavior, which is rarely available for devices today.

This paper presents the design and implementation of SUD, a kernel framework that provides complete isolation for *untrusted* device drivers in Linux, without the need for any special programming languages or specifications. SUD leverages recent hardware support to implement general-purpose mechanisms that ensure a misbehaving driver, and the hardware device that it manages, cannot compromise the rest of the system. SUD allows unmodified Linux device drivers to execute in untrusted user-space processes, thereby limiting the effects of bugs or security vulnerabilities in existing device drivers. SUD also ensures the safety of applications with direct access to hardware, such as the Linux X server, which today can corrupt the rest of the system.

Moving device drivers to untrusted user-space code in any system requires addressing three key challenges. First, many device drivers require access to privileged CPU instructions in order to interact with its device. However, to protect the rest of the system, we need to confine the driver's execution. Second, the hardware device needs to perform a range of low-level operations, such as reading and writing physical memory via DMA and signaling interrupts. However, to protect the rest of the system from operations that a malicious driver might request of the device, we must also control the operations that can be performed by the device. Finally, we would like to reuse existing driver code for untrusted device drivers. However, existing drivers rely on a range of kernel facilities not available in user-space applications making it difficult to reuse them as-is.

SUD's design addresses these challenges for Linux as follows. First, to isolate driver code, SUD uses existing Unix protection mechanisms to confine drivers, by running each driver in a separate process under a separate Unix user ID. To provide the device driver with access to its hardware device, the kernel provides direct access to memory-mapped device IO registers using page tables, and uses other *x*86 mechanisms to allow controlled access to the IO-space registers on the device. The kernel also provides special device files for safely managing device state that cannot be directly exposed at the hardware level.

Addressing the second challenge of confining the physical hardware managed by a driver is more difficult, due to the wide range of low-level operations that hardware devices can perform. SUD assumes that, unlike the device *driver*, the device *hardware* is trusted, and in particular, that it correctly implements the PCI express specification [25]. Given that assumption, SUD relies on an IOMMU [3, 17] and transaction filtering in PCI express bridges [25] to control the memory operations issued by the device under the control of the driver. SUD also relies on message-signaled interrupts [24] to route and mask interrupts from the device.

Finally, to support unmodified Linux device drivers, SUD emulates the kernel runtime environment in an untrusted user-space process. SUD relies on UML [9] to implement the bulk of the kernel facilities, and allows drivers to access the underlying devices by using SUD's direct hardware access mechanisms provided by the underlying kernel. The underlying kernel, in turn, exposes an upcall interface that allows the user-space process to provide functionality to the rest of the system by implementing the device driver API.

We have implemented a prototype of SUD for the Linux kernel, and have used it to run untrusted device drivers for Gigabit Ethernet cards, 802.11 wireless cards, sound cards, and USB host controllers and devices. SUD runs existing Linux device drivers without requiring any source-code modifications. A benchmark measuring streaming network throughput achieves the same performance with both in-kernel Linux drivers and the same drivers running in user-space with SUD, saturating a Gigabit Ethernet link, although SUD imposes an 8–30% CPU overhead. Our experiments suggest that SUD protects the system from malicious device drivers, even if the device driver attempts to issue arbitrary DMA requests and interrupts from its hardware device.

SUD provides complete isolation of untrusted user-space processes with access to arbitrary PCI devices, without relying on any specialized languages or specifications. SUD demonstrates how this support can be used to run unmodified Linux device drivers in an untrusted user-space process with CPU overheads comparable to other driver isolation techniques, and the same mechanisms can be used to safely run other applications that require direct access to hardware devices. Finally, we are hopeful that by making only minimal changes to the Linux kernel—two loadable kernel modules—SUD can finally put these research ideas to use in practice.

The rest of this paper is structured as follows. We first review related work in Section 2. We present the design of SUD in Section 3, and describe our implementation of SUD for the Linux kernel in Section 4. Section 5 evaluates the performance of our SUD prototype. Section 6 discusses the limitations of SUD and future work, and Section 7 concludes.

2 RELATED WORK

There is a significant amount of related work on improving the reliability, safety, and reuse of device drivers. The key focus of SUD is providing strong confinement of unmodified device drivers on Linux. In contrast, many prior systems have required adopting either a new OS kernel, new drivers, or new specifications for devices. On the other hand, techniques that focus on device driver reliability and reuse are complementary to SUD, and would apply equally well to SUD's untrusted drivers. The rest of this section surveys the key papers in this area.

Nooks [30] was one of the first systems to recognize that many kernel crashes are caused by faulty device drivers. Nooks used page table permissions to limit the effects of a buggy device driver's code on the rest of the kernel. However, Nooks was not able to protect the kernel from all possible bugs in a device driver, let alone malicious device driver code, whereas SUD can.

A few techniques for isolating kernel code at a finer granularity and with lower overheads than page tables have been proposed, including software fault isolation [6, 10] and Mondrian memory protection [34, 35]. While both SFI and MMP are helpful in confining the actions of driver code, they cannot confine the operations performed by the hardware device on behalf of the device driver, which is one of the key advantages of SUD.

Microkernels partition kernel services, including drivers, into separate processes or protection domains [2, 31]. Several microkernels, such as Minix 3 and L4, recently added support for IOMMU-based isolation of device DMA, which can prevent malicious device drivers from compromising the rest of the system [2, 15]. SUD borrows techniques from this previous work, but differs in that it aims to isolate unmodified Linux device drivers. We see this as a distinct challenge from previous work, because Linux device drivers are typically more complex than their microkernel counterparts[1] and SUD does not change the kernel-driver interface to be more amendable to isolation.

Virtual machine monitors must deal with similar issues to allow guest OS device drivers to directly access underlying hardware devices, and indeed virtualization is the key reason for the availability of IOMMU hardware, which has now been used in a number of VMMs [4, 23, 32]. In a virtual machine, however, malicious drivers can compromise their own guest OS and any applications the guest OS is running. SUD runs a separate UML process for each device driver; in this model, a driver compromising its user-space UML kernel is similar to a process compromising its libc. Thus, in SUD, the Linux kernel prevents a malicious driver from compromising other device drivers or applications.

[1]For example, the Linux e1000 Ethernet device driver is 13,000 lines of C code, and the Minix 3 e1000 driver is only 1,250 lines.

Loki [36] shows how device drivers, among other parts of kernel code, can be made untrusted by using physical memory tagging. However, Loki incurs a memory overhead for storing tags, and requires modifying both the CPU and the DMA engines to perform tag checks, which is unlikely to appear in mainstream systems in the near future. Unlike SUD, Loki's memory tagging also cannot protect against devices issuing arbitrary interrupts.

Many of the in-kernel isolation techniques, including Nooks, SFI, and MMP, allow restarting a crashed device driver. However, doing so requires being able to reclaim all resources allocated to that driver at runtime, such as kernel memory, threads, stacks, and so on. By running the entire driver in an untrusted user-level process, SUD avoids this problem altogether.

Another approach to confining operations made by the hardware device on behalf of the driver is to rely on a declarative specification of the hardware's state machine, such as in Nexus [33] or Termite [28]. These techniques can provide fine-grained safety properties specific to each device, using a software reference monitor to control a driver's interactions with a device. However, if a specification is not available, or is incorrect, such a system would not be able to confine a malicious device driver, since it is impossible to predict how interactions between the driver and the device translate into DMA accesses initiated by the device.

In contrast to a specification-based approach, SUD enforces a single safety specification, namely, memory safety for PCI express devices. It does so without relying on precise knowledge of when a device might issue DMA requests or interrupts, by using hardware to control device DMA and interrupts, and providing additional system calls to allow driver manipulation of PCI register state. The drawback of enforcing a single memory safety property is that SUD cannot protect physical devices from corruption by misbehaving drivers, unlike [33]. We expect that the two techniques could be combined, by enforcing a base memory safety property in SUD, and using finer-grained specifications to ensure higher-level properties.

Nooks introduced the concept of shadow drivers [29] to recover device driver state after a fault, and SUD's architecture could also use shadow drivers to gracefully restart untrusted device drivers. In SUD, shadow drivers could execute either in fully-trusted kernel code, or in a separate untrusted user-space process, isolated from the untrusted driver they are shadowing. Techniques from CuriOS [8] could likewise be applied to address this problem.

Device driver reuse is another important area of related work. Some of the approaches to this problem have been to run device drivers in a virtual machine [22] without security guarantees, or to synthesize device drivers from a common specification language [28]. By allowing untrusted device drivers to execute in user-space, SUD simplifies the task of reusing existing, unmodified device drivers safely across different kernels. A well-defined driver interface, such as [28], would make it easier to move drivers to user-space, but SUD's architecture would still provide isolation.

Even if a driver cannot crash the rest of the system, it may fail to function correctly. A number of systems have been developed to help programmers catch common programming mistakes in driver code [19, 27], to make sure that the driver cannot mis-configure the physical device [33], and to guarantee that the driver implements the hardware device's state machine correctly [28]. A well-meaning driver running under SUD would benefit from all of these techniques, but the correctness of these techniques, or whether they were used at all, would not impact the isolation guarantees made by SUD.

Finally, user-space device drivers [21] provide a number of well-known advantages over running drivers in the kernel, including ease of debugging, driver upgrades, driver reuse, and fault isolation from many of the bugs in the driver code. Microdrivers [12] shows that the performance-critical aspects of a driver can be moved into trusted kernel code, while running the bulk of the driver code in user-space with only a small performance penalty, even if written in a type-safe language like Java [26].

SUD achieves the same benefits of running drivers in user-space, but does not rely on any device-specific trusted kernel code. This allows SUD to run arbitrary device drivers and applications with direct hardware access, without having to trust any part of them ahead of time, at the cost of somewhat higher CPU overheads as compared to Microdrivers. We expect that performance techniques from other user-level device driver systems [21] can be applied to SUD to similarly reduce the CPU overhead.

3 DESIGN

The goal of SUD is to prevent a misbehaving device driver from corrupting or disabling the rest of the system, including the kernel, applications, and other drivers.[2] At the same time, SUD strives to provide good performance in the common case of well-behaved drivers. SUD assumes that the driver can issue arbitrary instructions or system calls, and can also configure the physical device to issue arbitrary DMA operations or interrupts. The driver can also refuse to respond to any requests, or simply go into

[2]Of course, if the application relies on the device in question, such as a web server relying on the network card, the application will not be able to make forward progress until a working device driver is available.

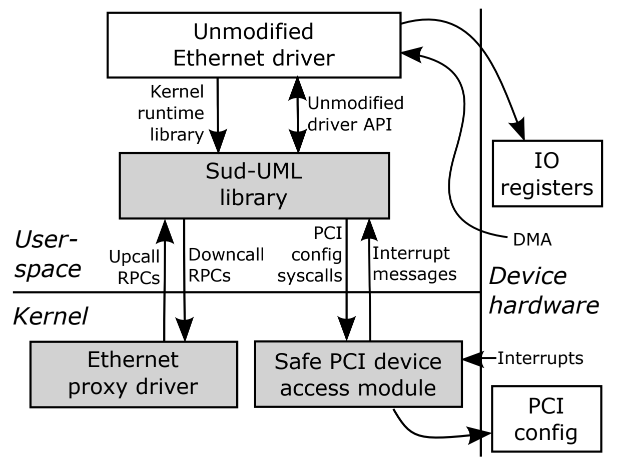


Figure 1: Overview of the interactions between components of SUD (shaded). Shown in user-space is an unmodified Ethernet device driver running on top of SUD-UML. A separate driver process runs for each device driver. Shown in kernel-space are two SUD kernel modules, an Ethernet proxy driver (used by all Ethernet device drivers in SUD), and a safe PCI device access module (used by all PCI card drivers in SUD). Arrows indicate request flow.

an infinite loop. To confine drivers, SUD assumes the use of recent *x*86 hardware, as we detail in Section 3.2.

The design of SUD consists of three distinct components, as illustrated in Figure 1. First, a *proxy driver* Linux kernel module allows user-space device drivers to implement the device driver interface expected by the Linux kernel. This kernel module acts as a proxy driver that can implement a particular type of device, such as an Ethernet interface or a wireless card. The proxy driver's job is to translate kernel calls to the proxy driver into upcalls to the user-space driver. A *safe PCI device access* kernel module allows user-space drivers to manage a physical hardware device, while ensuring that the driver cannot use the device to corrupt the rest of the system. Finally, a user-space library based on User-Mode Linux (UML) [9], called SUD-UML, allows unmodified Linux device drivers to run in untrusted user-space processes.

The rest of this section describes how the user-space device driver interacts with the rest of the system, focusing on how isolation is ensured for malicious device drivers.

3.1 API between kernel and driver

Traditional in-kernel device drivers interact with the kernel through well-known APIs. In Linux a driver typically registers itself with the kernel by calling a register function and passing a `struct` initialized with driver specific data and callbacks. The kernel invokes the callbacks to pass data and control to the driver. Likewise, drivers deliver data and execute kernel functions by calling predefined kernel functions.

As a concrete example, consider the kernel device driver for an imaginary "nic" Ethernet device, whose pseudo-code is shown in Figure 2. The Linux PCI

```
void nic_xmit_frame(struct sk_buff *skb)
{
    /*
     * Transmit a packet on behalf of the networking
     * stack.
     */
    nic_tx_buffer(skb->data, skb->data_len);
}

void nic_do_ioctl(int ioctl, char *result)
{
    /* Return MII media status. */
    if (ioctl == SIOCGMIIREG)
        nic_read_mii(result);
}

void nic_irq_handler(void)
{
    /*
     * Pass a recently received packet up to the
     * networking stack.
     */
    struct sk_buff *skb = nic_rx_skb();
    netif_rx(skb);
}

void nic_open(void)
{
    /*
     * Register an IRQ handler with the kernel and
     * enable the nic.
     */
    request_irq(nic_irq_num(),
                nic_irq_handler);
    nic_enable();
}

void nic_close(void)
{
    free_irq(nic_irq_num());
}

struct net_device_ops nic_netdev_ops = {
    .ndo_open       = nic_open,
    .ndo_stop       = nic_close,
    .ndo_start_xmit = nic_xmit_frame,
    .ndo_do_ioctl   = nic_do_ioctl,
};

int nic_probe(void)
{
    /* Register a device driver with the kernel. */
    char mac[6];
    struct net_device *netdev = alloc_etherdev();
    netdev->netdev_ops = &nic_netdev_ops;
    nic_read_mac_addr(mac);
    memcpy(netdev->dev_addr, mac, 6);
    register_netdev(netdev);
    return 0;
}
```

Figure 2: Example in-kernel Ethernet driver code, with five callback functions `nic_open`, `nic_close`, `nic_xmit_frame`, `nic_do_ioctl`, and `nic_irq_handler`. Ethernet drivers for real device require more lines of code.

code calls `nic_probe` when it notices a PCI device that matches the nic's PCI device and vendor ID. `nic_probe` allocates and initializes a `struct net_device` with the nic's MAC address and set of device specific callback functions then registers with Linux by calling `register_netdev`. When a user activates the device (*e.g.* by calling `ifconfig eth0 up`), Linux invokes the `nic_open` callback, which registers an IRQ handler to handle the device's interrupts, and enables the nic. When the Linux networking stack needs to send a packet, it passes the packet to the `nic_xmit_frame` callback. Likewise, when the nic receives a packet it passes the packet to the networking stack by calling `netif_rx`.

In order to move device drivers into user-space processes, SUD must translate the API between the kernel and the device driver, such as the example shown in Figure 2, into a message-based protocol that is more suitable for user-kernel communication. SUD uses proxy drivers for this purpose. A SUD proxy driver registers with the Linux device driver subsystem, providing any callback functions and data required by the class of drivers it supports. When the kernel invokes a proxy driver's callback function, the proxy driver translates the callback into an RPC call into a user-space driver. The kernel-driver API can also include shared memory that is accessed by either the driver or the kernel without invoking each other's functions. In our Ethernet driver example, the card's MAC address is stored in `netdev->dev_addr`, and is accessed without the use of any function calls. The proxy driver synchronizes such shared memory locations by mirroring, as we will discuss in Section 3.3.

SUD proxy drivers use a remote procedure call abstraction called *user channels* (or uchans for short), which we've optimized for messaging using memory shared between kernel and user address spaces. Figure 3 provides an overview of the SUD uchan interface. SUD implements uchans as special file descriptors. A uchan library translates the API in Figure 3 to operations on the file descriptor.

When the kernel invokes one of the proxy driver's callbacks, such as the function for transmitting a packet, the proxy driver marshals that request into an *upcall* into the user-space process. In the case of transmitting a packet, the proxy driver copies packet information into a `msg_t`, and calls `sud_asend` to add the `msg_t` to the queue holding kernel-to-user messages. Since transmitting a packet does not require an immediate reply, the proxy asynchronously sends the `msg_t`. On the other hand, synchronous upcalls are used for operations that require an immediate reply, such as `ioctl` calls to query the current MII media status of an Ethernet card, and result in the message being sent with `sud_send`, which blocks the callback until the user-space driver replies to the message.

kernel and user-space functions	
`sud_send(msg_t)`	Send a synchronous message.
`sud_asend(msg_t)`	Send an asynchronous message.
`buf_t sud_alloc()`	Allocate a shared buffer.
`sud_free(buf_t)`	Free a shared buffer.

user-space functions	
`msg_t sud_wait()`	Wait for a message.
`sud_reply(msg_t)`	Reply to a message.

Figure 3: Overview of the SUD uchan and memory allocation API.

The user-space process is responsible for handling kernel upcall messages, and typically the driver waits for a message from the kernel by calling `sud_wait`. When the proxy driver places on a message on the kernel-to-user queue, `sud_wait` dequeues the message and returns it to the user-space driver. The user-space driver processes the message by unmarshaling the arguments from the message, and invoking the corresponding callback in the driver code. If the callback returns a result (*i.e.* the kernel called `sud_send`), the user-space driver marshals the response into a `msg_t`, and calls `sud_reply`, which places the `msg_t` on a queue holding user-to-kernel messages.

When the user-to-kernel message queue contains a reply message, the proxy driver removes the message from the queue and unblocks the callback waiting for the reply. The callback completes by returning appropriate results to its caller. In the `ioctl` example, the kernel passes a buffer to the callback that the callback copies the result from the user-space driver reply into.

User-space drivers may also need to invoke certain kernel functions, such as changing the link status of the Ethernet interface. This is called a *downcall* in SUD, and is implemented in an analogous fashion, where the user-space driver and the in-kernel proxy driver reverse roles in the RPC protocol. One notable difference is that the kernel returns results of downcalls directly by copying the results into the message buffer the driver passed to `sud_send`, instead of by sending a separate message to the driver process.

3.1.1 *Protecting the kernel from the device driver*

Moving device drivers to user-space prevents device drivers from accessing kernel memory directly. This prevents buggy or malicious device drivers from crashing the kernel. However, a buggy or malicious user-space device driver may still break the kernel, other processes, or other devices, unless special precautions are taken at the user-kernel API. The kernel, and the proxy driver in particular, needs to make as few assumptions as possible about the behavior of the user-space device driver. The rest of this subsection describes how SUD handles liveness and semantic assumptions.

Liveness assumptions. One assumption that is often made of trusted in-kernel drivers is that they will handle requests in a timely fashion. However, if a malicious user-space device driver fails to respond to upcalls, many threads in the kernel may eventually be blocked waiting for responses that will never arrive. SUD addresses this problem in two ways. First, for upcalls that require a response from the user-space device driver before the in-kernel proxy can proceed, the upcall is made *interruptable*. This allows the user to abort (`Ctrl-C`) an operation that appears to have hung, such as running `ifconfig` on an unresponsive driver. To implement interruptable upcalls, the user-kernel interface must be carefully designed to allow any synchronous upcall to return an error.

Second, SUD uses *asynchronous* upcalls whenever possible. Asynchronous upcalls can be used in situations where the in-kernel proxy driver does not require any response from the user-space driver in order to return to its caller safely, such as packet transmission. If the device driver's queue is full, the kernel can wait a short period of time to determine if the user-space driver is making any progress at all, and if not, the driver can be reported as hung to the user.

Asynchronous upcalls are also necessary for handling upcalls from threads running in a non-preemptable context, such as when holding a spinlock. A thread in a non-preemptable context cannot go to sleep, and therefore cannot allow the user-space driver to execute and process the upcall. While multi-core systems can avoid this problem by running the driver and the non-preemptable context concurrently, SUD still must not rely on the liveness of the untrusted device driver.

A potential problem can occur if a non-preemptable kernel thread invokes the in-kernel proxy driver and expects a response (so that the proxy driver might need to perform an upcall). One solution to this problem is rewriting the kernel code so a non-preemptable context is unnecessary. In Linux, for example, we could replace the spin lock with a mutex. However, this solution is undesirable, because it might require restructuring portions of the kernel and affect performance poorly.

To address this problem, we observe that the work performed by functions called in a non-preemptable context is usually small and well-defined; after all, the kernel tries to avoid doing large amounts of work in a non-preemptable context. Thus, for every class of devices, the corresponding SUD proxy driver implements any short functions invoked by the kernel as part of the driver API.[3] Any state required by these functions is mirrored and synchronized between the real kernel and the SUD-UML kernel. For example, the Linux 802.11 network stack calls the driver to enable certain features, while executing in a non-preemptable context; the driver must respond with the features it supports and will enable. The wireless proxy driver mirrors the (static) supported feature set, and when the kernel invokes the function to enable some feature, the proxy driver queues an asynchronous upcall to SUD-UML containing the newly-enabled features.

[3]While we have found that this approach works for device drivers we have considered so far, it is possible that other kernel APIs have long, device-specific functions invoked in a non-preemptable context. Supporting these devices in SUD would require modifying the kernel, as Section 3.1.3 discusses.

Semantic assumptions. A second class of assumptions that kernel code may make about trusted in-kernel drivers has to do with the *semantics* of the driver's responses. A hypothetical kernel might rely on the fact that, once the kernel changes the MAC address of an Ethernet card, a query to get the current MAC address will return the new value. SUD does not enforce such invariants on drivers, because we have not found any examples of such assumptions in practice. Infact, Linux subsystems that interact with device drivers (such as the network device subsystem) are often robust to driver mistakes, and print error messages when the driver is acting in unexpected ways. At this point, the administrator can kill the misbehaving user-space driver. If the kernel did rely on higher-level invariants about driver behavior, the corresponding proxy driver would need to be modified to keep track of the necessary state, and to enforce the invariant.

3.1.2 Uchan optimizations

Two potential sources of performance overhead in SUD come from the context switches due to upcalls and downcalls, and from data copying, such as the packets in an Ethernet driver.

The SUD uchan implementation optimizes the number of context switches due to upcalls and downcalls by mapping message queues into memory shared by the kernel and user-space driver. SUD uchans implement the message queues using ring buffers. The kernel writes messages into the head of the kernel-to-user ring. When the user-space driver calls `sud_wait` to wait for a message, `sud_wait` polls the kernel-to-user ring tail pointer. If the tail points to a message `sud_wait` dequeues the message by incrementing the tail pointer, the user-space driver processes the message, and possibly returns results by calling `sud_reply`. When the tail of the ring equals the head, the queue is empty and the user-space process sleeps by calling `select` on the uchan file descriptor. `select` returns when the kernel adds a message to the head of the kernel-to-user ring. This interface allows a user-space process to process multiple messages without entering the kernel.

The downcall message queues work in a similar fashion, except that the user-space driver writes to the head of the ring and the kernel reads from the tail of the ring. When a user-space driver calls `sud_asend`, the uchan library adds the message to the queue, but does not notify the kernel

of the pending message until the user-space driver calls `sud_wait` or `sud_send`. This allows user-space drivers to batch asynchronous downcalls.

SUD also optimizes data copying overhead by pre-allocating data buffers in the user-space driver, and having the in-kernel proxy driver map them in the kernel's address space. A call to `sud_alloc` returns one of the shared messages buffers and `sud_free` returns the message buffer to the shared heap. In an Ethernet driver, this allows packet transmit upcalls and packet receive downcalls to exchange pointers using `sud_send`, and avoid copying the data. As we will describe later, the same shared buffers are passed to the physical device to access via DMA, avoiding any additional data copy operations in the user-space driver.

The in-kernel proxy driver may need to perform one data copy operation to guard against malicious user-space drivers changing the shared-memory data after it has been passed to the kernel. For example, a malicious driver may construct an incoming packet to initially look like a safe packet that passes through the firewall, but once the firewall approves the packet, the malicious driver changes the packet in shared memory to instead connect to a firewalled service. In the case of network drivers, we can avoid the overhead of this additional data copy operation by performing it at the same time that the packet's checksum is computed and verified, at which point the data is already being brought into the CPU's data cache. An alternative design may be to mark the page table entries read-only, but we have found that invalidating TLB entries from the IOMMU's page table is prohibitively expensive on current hardware.

3.1.3 Limitations

The implementation of the Linux kernel imposes several limitations on what types of device drivers SUD supports and what driver features a proxy driver can support.

It is unlikely SUD will ever be able to support device drivers that are critical for a kernel to function. For example, Linux relies on a functioning timer device driver to signal the scheduler when a time quantum elapses. A buggy or malicious timer driver could deadlock the kernel, even while running as a SUD user-space driver.

Another limitation is how proxy drivers handle callbacks when the calling kernel thread is non-preemptable. Servicing the callback in the in-kernel proxy allow SUD to support common functions for several device classes, but it does not work in all cases. For SUD to support all the functions of Linux kernel devices it is likely that some kernel subsystems would need to be restructured so non-preemptable threads do not need to make upcalls.

Despite these limitations SUD supports common features for several widely used devices. We could incrementally add support for more functions.

3.2 Confining hardware device access

The key challenge to isolating an untrusted device driver is making sure that a malicious driver cannot misuse its access to the underlying hardware device to escape isolation. In this subsection, we discuss how SUD confines the driver's interactions with the physical device, first focusing on operations that the driver can perform on the device, and second discussing the operations that the device itself may be able to perform.

To control access to devices without knowing the details of the specific device hardware interface, SUD assumes that all devices managed from user-space are PCI devices. This assumption holds for almost all devices of interest today.

3.2.1 Driver-initiated operations

In order for the user-space device driver to function, it must be able to perform certain operations on the hardware device. This includes accessing the device's memory-mapped IO registers, accessing legacy *x*86 IO registers on the device, and accessing the device's PCI configuration registers. SUD's safe PCI device access module, shown in Figure 1, is responsible for supporting these operations.

To allow access to memory-mapped IO registers, SUD's PCI device access module allows a user-space device driver to directly map them into the driver's address space. To make sure that these page mappings do not grant unintended privileges to an untrusted device driver, SUD makes sure that all memory-mapped IO ranges are page-aligned, and does not allow untrusted drivers to access pages that contain memory-mapped IO registers from multiple devices.

Certain devices and drivers also require the use of legacy *x*86 IO-space registers for initialization. To allow drivers to access the device's IO-space registers, SUD uses the IOPB bitmask in the task's TSS [16] to permit access to specific IO ports.

Finally, drivers need to access the PCI configuration space of their device to set certain PCI-specific parameters. However, some of the PCI configuration space parameters might allow a malicious driver to intercept writes to arbitrary physical addresses or IO ports, or issue PCI transactions on behalf of other devices. To prevent such attacks, SUD exposes PCI configuration space access through a special system call interface, instead of granting direct hardware access to the user-space driver. This allows SUD to ensure that sensitive PCI configuration registers are not modified by the untrusted driver.

3.2.2 Device-initiated operations

A malicious user-space driver may be able to escape isolation by asking the physical hardware device to perform operations on its behalf, such as reading or writing physical memory via DMA, or raising arbitrary interrupts. To prevent such problems, SUD uses hardware mechanisms, as shown in Figure 4, to confine each physical device managed by an untrusted device driver.

DMA. First and foremost, SUD must ensure that the device does not access arbitrary physical memory via DMA. To do so, SUD relies on IOMMU hardware available in recent Intel [17] and AMD [3] CPUs and chipsets to interpose on all DMA operations. The PCI device access module specifies a page table for each PCI device in the system, and the IOMMU translates the addresses in each DMA request according to the page table for the originating PCI device, much like the CPU's MMU. By only inserting page mappings for physical pages accessible to the untrusted driver into the corresponding PCI device's IO page table, SUD ensures that a PCI device cannot access any physical memory not already accessible to the untrusted driver itself.

Peer-to-peer DMA. Although an IOMMU protects the physical memory of the system from device DMA requests, a subtle problem remains. Traditional PCI bridges route DMA transactions according to the destination physical address, and a PCI device under the control of a malicious driver may be able to DMA into the memory-mapped registers of another PCI device managed by a different driver. As can be seen in Figure 4, a DMA transaction from device A to the physical address of device B's registers would not cross the IOMMU, and thereby would not be prevented.

To avoid this problem, SUD requires the use of a PCI express bus, which uses point-to-point physical links between PCI devices and switches, as opposed to traditional PCI which uses a real bus shared by multiple devices. When multiple devices share the same physical PCI bus, there is nothing that can prevent a device-to-device DMA attack. With PCI express, at least one PCI express switch is present between any two devices, and can help us avoid this problem.

To ensure that all PCI requests pass through the root switch, SUD enables PCI access control services (ACS) [25] on all PCI express switches. ACS allows the operating system to control the routing and filtering of certain PCI requests. In particular, SUD enables source validation, which ensures that a downstream PCI device cannot spoof its source address, and P2P request and completion redirection, which ensures that all DMA requests and responses are always propagated from devices to the root (where the IOMMU is located), and from the root to the devices, but never from one device to another.

Interrupts. The final issue that SUD must address is interrupts that can be raised by devices. Although interrupts are unlikely to corrupt any system state on their own, a malicious driver could use an interrupt storm to force CPUs to keep handling interrupts, thereby livelocking the system. Traditionally, the device driver's interrupt handler is responsible for clearing the interrupt condition before interrupts are re-enabled. In some cases devices share an interrupt and are expected to coordinate interrupt handling. With untrusted device drivers, however, the kernel cannot rely on a driver to clear the interrupt condition or cooperate with other drivers, so SUD takes a few measures to prevent this from happening.

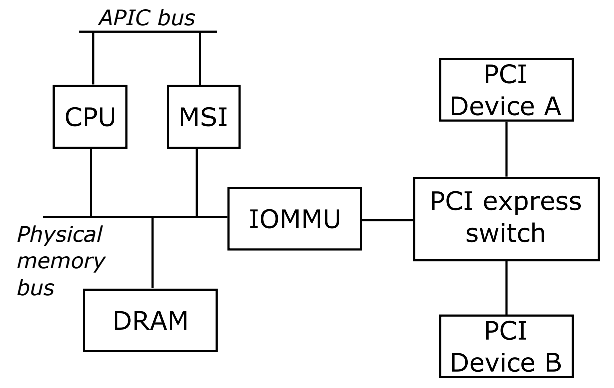


Figure 4: Overview of the hardware mechanisms used by SUD to confine hardware devices managed by untrusted drivers. In some systems, the APIC bus is overlaid on the physical memory bus, and in some systems the DRAM is attached directly to the CPU.

First, SUD prevents devices from raising legacy interrupts shared by multiple devices. SUD does so by restricting drivers from directly accessing the PCI configuration registers to change the interrupt configuration. Instead of legacy interrupts, SUD relies on message-signaled interrupts (MSI), which are mandatory for PCI express devices, and support generic interrupt masking that does not depend on the specific device.

Second, SUD forwards device interrupts to untrusted drivers via the upcall mechanism that was described in Section 3.1. When an interrupt comes in, SUD issues an upcall to the corresponding driver indicating that an interrupt was signaled. At this point, SUD does not mask further MSI interrupts, since they are edge-triggered. However, if another interrupt for the same device comes in, before the driver indicates that it has finished processing the interrupt, SUD uses MSI to make sure further interrupts do not prevent the driver's forward progress.

Note that this design allows the OS scheduler to schedule device drivers along with other processes in the system. When the device driver's time quantum expires, it will be descheduled, and even if it were handling an interrupt, MSI will be used to mask subsequent interrupts until the driver can run again. Of course, in practice it may be desirable to run device drivers with high priority, to make sure that devices are serviced promptly, but should a driver misbehave, other processes will still be able to execute.

The final consideration has to do with how message-signaled interrupts are implemented on *x*86. A device signals an MSI by performing a memory write to a reserved physical memory address, and the MSI controller on the other side of the IOMMU picks up writes to that memory address and translates them into CPU interrupts on the APIC bus, as shown in Figure 4. Unfortunately, it is impossible to determine whether a write to the MSI address was caused by a real interrupt, or a stray DMA write to the same address. SUD can mask real interrupts by changing the MSI register in the PCI configuration space of the device, but cannot prevent stray DMA writes to the MSI address.

To avoid interrupt storms and arbitrary interrupts caused by a malicious driver using DMA to the MSI address, SUD uses two strategies, depending on which IOMMU hardware it has available to it. On Intel's VT-d [17], SUD uses interrupt remapping support. Interrupt remapping allows the OS kernel to supply a table translating potential MSI interrupts raised by each device to the physical interrupt that should be raised as a result, if any. Changing an interrupt remapping table is more expensive than using MSI masking, so SUD first tries to use MSI to mask an interrupt, and if that fails, SUD changes the interrupt remapping table to disable MSI interrupts from that device altogether. With AMD's IOMMU [3], SUD removes the mapping for the MSI address from that particular device's IO page table, thereby preventing that device from performing any MSI writes.

3.3 Running unmodified drivers in user-space

To allow unmodified Linux device drivers to run in untrusted user-space processes, SUD uses UML [9] to supply a kernel-like runtime environment in user-space, which we call SUD-UML. SUD-UML's UML environment provides unmodified drivers with all of the kernel code they rely on, such as `kmalloc()` and `jiffies`.

SUD-UML differs from traditional UML in three key areas that allow it to connect drivers to the rest of the world. First, SUD-UML replaces low-level routines that access PCI devices and allocate DMA memory with calls to the underlying kernel's safe PCI device access module, shown in Figure 1. For example, when the user-space driver calls `pci_enable_device` to enable a PCI device, SUD-UML performs a downcall to the underlying kernel to do so. When the user-space driver allocates DMA-capable memory, SUD-UML requests that the newly-allocated memory be added to the IOMMU page table for the relevant device. When the driver registers an interrupt handler, SUD-UML asks the underlying kernel to forward interrupt upcalls to it.

Second, SUD-UML implements the user-kernel RPC interface that we described in Section 3.1 by invoking the unmodified Linux driver when it receives an upcall from the kernel, and sends the response, if any, back to the kernel over the same file descriptor. For example, when an interrupt upcall is received by SUD-UML, it invokes the interrupt handler that was registered by the user-space driver.

Finally, SUD-UML mirrors shared-memory state that is part of the driver's kernel API, by maintaining the same state in both the real kernel and SUD-UML's UML kernel, and synchronizing the two copies as needed. For example, the Linux kernel uses shared memory variables to track the link state of an Ethernet interface, or the currently available bitrates for a wireless card. The SUD proxy driver and SUD-UML cooperate to synchronize the two copies of the state. If the proxy driver updates the kernel's copy of the state, it sends an upcall to SUD-UML with the new value. In SUD-UML, we exploit the fact that updates to driver shared memory variables are done via macros, and modify these macros to send a downcall with the new state to the real kernel. This allows the user-space device driver to remain unchanged.

Updates to shared-memory state are ordered with respect to all other upcall and downcall messages, which avoids race conditions. Typically, any given shared-memory variable is updated by either the device driver or by the kernel, but not both. As a result, changes to shared-memory state appear in the correct order with respect to other calls to or from the device driver. However, for security purposes, the only state that matters is the state in the real kernel. As discussed in Section 3.1.1, the Linux kernel is robust with respect to semantic assumptions about values reported by device drivers.

Poorly written or legacy drivers often fail to follow kernel conventions for using system resources. For example, some graphics cards set up DMA descriptors with physical addresses instead of using the kernel DMA interface to get a DMA capable address. A poorly written driver will run in SUD-UML until it attempts to accesses a resource that it has not properly allocated. When this happens the SUD-UML process terminates with an error.

4 IMPLEMENTATION

We have implemented a prototype of SUD as a kernel module for Ubuntu's Linux 2.6.30-5 kernel on x86-64 CPUs. We made several minor modifications to the Linux kernel proper. In particular, we augmented the DMA mapping interface to include functions for flushing the IOTLB, mapping memory starting at a specified IO virtual address, and garbage collecting an IO address space. We have only tested SUD on Intel hardware with VT-d [17], but the implementation does not rely on VT-d features, and SUD should run on any IOMMU hardware that provides DMA

Feature	Lines of code
Safe PCI device access module	2800
Ethernet proxy driver	300
Wireless proxy driver	600
Audio card proxy driver	550
USB host proxy driver	0
SUD-UML runtime	5000

Figure 5: Lines of code required to implement our prototype of SUD, SUD-UML, and each of the device-specific proxy drivers. SUD-UML uses about 3 Mbytes of RAM, not including UML kernel text, for each driver process.

address translation, such as AMD's IOMMU [3]. SUD-UML, our modified version of UML, is also based on Ubuntu's Linux 2.6.30-5 kernel.

Our current prototypes of SUD and SUD-UML include proxy drivers and UML support for Ethernet cards, wireless cards, sound cards, and USB host controllers and devices. On top of this, we have been able to run a range of device drivers as untrusted user-space processes, including the e1000e Gigabit Ethernet card driver, the iwlagn5000 802.11 wireless card driver, the ne2k-pci Ethernet card driver, the snd_hda_intel sound card driver, the EHCI and UHCI USB host controller drivers, and various USB device drivers, all with no modifications to the driver itself.

Figure 5 summarizes the lines of code necessary to implement the base SUD system, the base SUD-UML environment, and to add each class of devices. The USB host driver class requires no code beyond what is provided by the SUD core. Some USB devices, however, require additional driver classes. For example, a USB 802.11 wireless adapter can use the existing wireless proxy driver, and a USB sound card could use the audio card proxy driver. We are working on a block device proxy driver to support USB storage devices.

4.1 User-mode driver API

SUD's kernel module exports four SUD device files for each PCI device that it manages, as shown in Figure 6. The device files are initially owned by root, but the system administrator can set the owner of these devices to any other UID, and then run an untrusted device driver for this device under that UID.

When the system administrator starts a driver, SUD-UML searches `sysfs` for a matching device. If SUD-UML finds a matching device, it invokes the kernel to start a proxy driver and open a uchan shared with the proxy driver.

SUD-UML translates Linux kernel device driver API calls to operations on the SUD device files. When a device driver calls `dma_alloc_coherent`, SUD-UML uses `mmap` to allocate anonymous memory from the `dma_coherent` device. This allocates the requested number of memory pages in the driver's process, and also maps the same pages at the same virtual address in the corresponding device's IOMMU page table. Likewise, SUD-UML allocates cacheable DMA memory using anonymous `mmap` on `dma_caching`. The `mmio` file exports the PCI device's memory-mapped IO registers, which the driver accesses by `mmap`ing this device. Finally, The `ctl` file is used to handle kernel upcalls and to issue downcalls. Figure 7 gives sample of upcalls and downcalls.

System administrators can manage user-space drivers in the same way they manage other processes and daemons. An administrator can terminate a misbehaving or buggy driver with `kill -9`, and restart it by starting a new SUD-UML process for the device. The Linux functions for managing resource limits, such as `setrlimit`, work for user-space drivers. For example, an administrator might use `setrlimit` to limit the amount of memory a suspicious driver is allowed to allocate.

Some device drivers, such as sound card drivers, might require real time scheduling constraints to function properly, for example to ensure a high bit rate. For these devices an administrator can use `sched_setscheduler` to assign the user-space driver one of Linux' real-time scheduling policies. If the audio driver turns out to be malicious or buggy, it could consume a large fraction of CPU time, but unlike a fully-trusted kernel driver, it would not be able to lock up the system entirely.

4.2 Performance optimizations

Most of the performance overhead in SUD comes from SUD-UML. Our efforts to optimize SUD-UML are ongoing, but we have implemented several important optimizations, which we describe in the following paragraphs.

One optimization is to handle upcalls and invoke callbacks directly from the UML idle thread. This must be done with care, however, because some drivers implement callbacks that block the calling thread, but expect other threads to continue to invoke driver callbacks. For example, the e1000e driver determines which type of interrupt to configure (*e.g.* legacy or MSI) by triggering the interrupt, sleeping, and then checking a bit that should have been set by the e1000e interrupt handler.

To handle this case, when the UML idle thread receives an upcall, it checks if the corresponding callback is allowed to block (according to kernel conventions). If the callback is not allowed to block, the UML idle thread invokes the callback directly. Otherwise, the idle thread creates and runs a worker thread to invoke the callback. We optimize worker thread creation using a thread pool.

Another optimization, which we have not implemented yet, but we expect to improve performance, it to use superpages to map SUD-UML's memory, including memory shared with the kernel. The kernel must flush all non-kernel mappings when it performs a context switch between user-space virtual address spaces. This impacts

File	Use
`/sys/devices/.../sud/ctl`	Transfers upcall and downcall messages.
`/sys/devices/.../sud/mmio`	Represents the PCI device's memory-mapped IO regions; intended for use with `mmap`.
`/sys/devices/.../sud/dma_coherent`	Allocates anonymous non-caching memory on `mmap`, mapped at the same virtual address in both the driver's page table, and the device's IOMMU page table.
`/sys/devices/.../sud/dma_caching`	Allocates anonymous caching memory on `mmap`, mapped at the same virtual address in both the driver's page table, and the device's IOMMU page table.

Figure 6: An overview of device files that SUD exports for each PCI device.

Upcall	Description
`ioctl`	Request that the driver perform a device-specific `ioctl`.
`interrupt`	Invoke the SUD-UML driver interrupt handler.
`net_open`	Prepare a network device for operation.
`bss_change`	Notify an 802.11 device that the BSS has changed.

Downcall	Description
`interrupt_ack`	Request that SUD unmask the device interrupt line.
`request_region`	Add IO-space ports to the driver's IO permission bitmask.
`netif_rx`	Submit a received packet to the kernel's network stack.
`pci_find_capability`	Checks if device supports a particular capability.

Figure 7: A sample of SUD upcalls and downcalls.

user-space drivers, because the drivers often have a large working set of DMA buffers. For example, the e1000e allocates 256 buffers, which might span multiple pages, for both the transmit and receive DMA rings. In the case of the e1000e driver, the driver might read the contents of the DMA buffer after a packet has been received. This results in a TLB miss if the kernel context-switched from the SUD-UML process to another process since the last time the driver read the DMA buffer. By mapping all the DMA buffers using several super pages, SUD-UML could avoid many of these TLB misses.

5 EVALUATION

To evaluate SUD, we wanted to understand how hard it is to use SUD, how well it performs at runtime, and how well it protects the system from malicious drivers. The implementation section already illustrated that SUD allows existing Linux device drivers to be used as untrusted user-space drivers with no source-code modifications. In this section, we focus on the runtime overheads imposed by SUD for running device drivers in user-space, and on the isolation guarantees that SUD provides.

In short, our results show that SUD incurs performance overheads comparable to other device driver isolation techniques, while providing stronger guarantees. To illustrate SUD's security guarantees, we have verified that SUD prevents both DMA and interrupt attacks, as well as the driver's attempts to mishandle kernel upcalls. The rest of this section describes our experimental evaluation in more detail.

5.1 Network driver performance

The primary performance concern in running device drivers as user-space processes is the overhead of context switching and copying data. To understand the performance overhead imposed by SUD, we consider the *worst-case* scenario—a Gigabit Ethernet device that requires both high throughput and low latency to achieve high performance, using both small and large packets—so that any overhead introduced by SUD's protection mechanisms will show up clearly in the benchmark results. In practice, we expect most of the drivers running under SUD to be less performance-critical (for example keyboard, mouse, printer, or sound card drivers), and thus any performance penalties imposed by SUD on those drivers would be even less noticeable.

We run four netperf [18] benchmarks to exercise the e1000e Linux device driver running under SUD on an Thinkpad X301 with a 1.4GHz dual-core Intel Centrino. The Thinkpad is connected to a 2.8GHz dual-core Pentium D Dell Optiplex by a Gigabit switched network. We configure netperf to run experiments to report results accurate to 5% with 99% confidence.

Figure 8 summarizes performance results and CPU overheads for the untrusted driver running in SUD and the trusted driver running in the kernel. The first benchmark, `TCP_STREAM`, measures TCP receive throughput and is run with 87380 byte receive buffers and 16384 byte send buffers. SUD offers the same performance as the kernel driver with little overhead, because SUD-UML is able to batch delivery of many large packets to the kernel in one downcall.

The `UDP_STREAM` TX and RX benchmarks measure throughput for transmitting and receiving 64 byte UDP packets. These benchmarks are more CPU intensive than `TCP_STREAM`, and demonstrate overheads in SUD that might have been obscured by the use of large packets in `TCP_STREAM`. For both TX and RX, SUD performs comparably to the kernel driver, but has about a 11% overhead for TX and a 30% overhead for RX.

The final benchmark, `UDP_RR`, is designed to measure driver latency. The `UDP_RR` results are given in transactions per second. The client completes a transaction when it sends a 64 byte UDP packet and waits for the server to reply with a 64 byte UDP packet. In some ways this is a worst case benchmark for SUD, which has a CPU overhead of 2x. After each packet transmit or receive the e1000e process sleeps to wait for the next event. Unfortunately, waking up the sleeping process can take as long as $4\mu s$ in Linux. If e1000e driver has more work and sleeps less, such as in the `UDP_STREAM` benchmark, this high wakeup overhead is avoided.

5.2 Security

We argued in Section 3 that SUD uses IOMMUs to prevent devices from accessing arbitrary physical memory. Figure 9 shows the IO virtual memory mappings for the e1000e driver. We read all mappings by walking the e1000e device's IO page directory. This ensures that the BIOS or other system software does not create special mappings for device use. The lack of any other mappings indicates that a malicious device driver can at most corrupt its own transmit and receive buffers, or raise an interrupt using MSI.

Our experimental machine does not have support for interrupt remapping in its IOMMU hardware, so our configuration is vulnerable to livelock by a malicious driver issuing DMA requests to the MSI address. Unfortunately, Intel VT-d always includes an implicit identity mapping for the MSI address in every page table, so it was not possible to prevent this type of attack. A newer chipset version would have avoided this weakness, and we expect that doing so would not impact the performance of SUD. Alternatively, AMD's IOMMU does not include an implicit MSI mapping, and we could simply unmap the MSI address on an AMD system when an interrupt storm is detected, to prevent further interrupts from a device.

We tested SUD's security by constructing explicit test cases for the attacks we described earlier in Section 3, including arbitrary DMA memory accesses from the device and interrupt storms. In all cases, SUD safely confined the device and the driver. We have also relied on SUD's security guarantees while developing SUD-UML and testing drivers. For example, in one situation a bug in our SUD-UML DMA code was returning an incorrect DMA address, which caused the USB host controller driver to attempt a DMA to an unmapped address. The bug, however, was easy to spot, because it triggered a page fault. As another example, the SUD-UML interrupt code responsible for handling upcalls was not invoking the iwlagn5000 interrupt handler, but was re-enabling interrupts with SUD. The resulting interrupt storm was easily fixed by killing the SUD-UML process. It is also relatively simple to restart a crashed device driver by restarting the device driver process.

Test	Driver	Throughput	CPU %
`TCP_STREAM`	Kernel driver	941 Mbits/sec	12%
	Untrusted driver	941 Mbits/sec	13%
`UDP_STREAM TX`	Kernel driver	317 Kpackets/sec	35%
	Untrusted driver	308 Kpackets/sec	39%
`UDP_STREAM RX`	Kernel driver	238 Kpackets/sec	20%
	Untrusted driver	235 Kpackets/sec	26%
`UDP_RR`	Kernel driver	9590 Tx/sec	5%
	Untrusted driver	9489 Tx/sec	10%

Figure 8: TCP streaming, minimum-size UDP packet streaming, and UDP request-response performance for the e1000e Ethernet driver running as an in-kernel driver and as an untrusted SUD driver. Each UDP packet is 64-bytes.

Memory use	Start	End
TX ring descriptor	0x42430000	0x42431000
RX ring descriptor	0x42431000	0x42433000
TX buffers	0x42433000	0x42C33000
RX buffers	0x42C33000	0x43433000
Implicit MSI mapping	0xFEE00000	0xFEF00000

Figure 9: The IO virtual memory mappings for the e1000e driver.

6 Discussion

We think SUD demonstrates that unmodified device drivers can be run as user-space processes with good performance. This section examines some limitations of SUD and explores directions of future work.

New hardware. Our test machine does not support interrupt remapping, which leaves SUD vulnerable to a livelock attack from a malicious driver. The ability to remap interrupts is necessary to prevent this attack, but could also be useful for improving performance. For example, it might be faster to mask an interrupt by remapping the MSI page instead of by reconfiguring the PCI device.

Hardware queued IOTLB invalidation, which is present in some Intel VT-d implementations, allows software to queue several IOTLB invalidations efficiently. SUD could use this feature to unmap DMA buffers from the user-space device driver while they are being processed by the kernel.

Device delegation. In the current SUD design, the kernel defines all of the devices in the system (e.g., all PCI devices), and grants user-space drivers access at that granularity (e.g., one PCI device). An alternative approach that we hope to explore in the future is to allow one untrusted device driver to create new device objects, which could then be delegated to separate device driver processes. For example, the system administrator might start a PCI express bus process, which would scan the PCI express bus and start a separate driver process for each device it found. If one of the devices on the PCI express bus was a USB host controller, the USB host controller driver might start a new driver process for each USB device it found. If the device was a SATA controller, the SATA driver may likewise start a new driver for each disk.

Finally, a network card with hardware support for multiple virtual queues, such as the Intel IXGBE, could give applications direct access to one of its queues.

Optimized drivers. Supporting unmodified device drivers is a primary goal SUD-UML. However, porting drivers to a SUD interface might eliminate some CPU overhead that results from supporting unmodified drivers. For example, SUD-UML constructs Linux socket buffers for each packet the kernel transmits, because this is what the unmodified device expects. By modifying device drivers to take advantage of the SUD interface directly, we may be able to achieve lower CPU overheads as in [21].

Applications. There are some applications that are not necessarily suitable to run in the kernel, but that benefit from direct access to hardware. These applications either make do with sub-optimal performance, or are implemented as trusted modules and run in the kernel, in spite of the security concerns. For example, the Click [20] router runs as a kernel module so that it has direct access to packets as they are received by the network card. With SUD, these applications could run as untrusted SUD-UML driver processes, with direct access to hardware, and achieve good performance without the security threat.

7 Conclusion

SUD is a new system for confining buggy or malicious Linux device drivers. SUD confines malicious drivers by running them in untrusted user-space processes. To ensure that hardware devices controlled by untrusted drivers do not compromise the rest of the system through DMA or interrupt attacks, SUD uses IOMMU hardware, PCI express switches, and message-signaled interrupts. SUD can run untrusted drivers for arbitrary PCI devices, without requiring any specialized language or specification. Our SUD prototype demonstrates support for Ethernet cards, wireless cards, sound cards, and USB host controllers, and achieves performance equal to in-kernel drivers with reasonable CPU overhead, while providing strong isolation from malicious drivers. SUD requires minimal changes to the Linux kernel—a total of two kernel modules comprising less than 4,000 lines of code—which may finally help these research ideas to be applied in practice.

Acknowledgments

We thank Austin Clements and M. Frans Kaashoek for their feedback on this paper. Thanks to the anonymous reviewers, and to our shepherd, Jaeyeon Jung, for helping improve this paper as well.

References

[1] Linux kernel i915 driver memory corruption vulnerability. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3831`.

[2] The L4Ka Project. `http://l4ka.org/`.

[3] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification*, February 2006.

[4] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. Utilizing IOMMUs for virtualization in linux and xen. In *Proceedings of the2006 Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[5] L. Butti and J. Tinnes. Discovering and exploiting 802.11 wireless driver vulnerabilities. *Journal in Computer Virology*, 4(1), Feburary 2008.

[6] M. Castro, M. Costa, J. P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[7] P. Chubb. Linux kernel infrastructure for user-level device drivers. In *In Linux Conference*, 2004.

[8] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[9] J. Dike. The user-mode Linux kernel home page. `http://user-mode-linux.sf.net/`.

[10] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, November 2006.

[11] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX Large Installation System Administration Conference*, Washington, DC, December 2006.

[12] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, and S. Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, Seattle, WA, March 2008.

[13] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[14] H. Härtig, J. Loeser, F. Mehnert, L. Reuther, M. Pohlack, and A. Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08, TU Dresden, Dresden, Germany, July 2003.

[15] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the2009 IEEE Dependable Systems and Networks Conference*, Lisbon, Portugal, June–July 2009.

[16] Intel. *Intel 64 and IA-32 Architectures Developer's Manual*, November 2008.

[17] Intel. Intel Virtualization Technology for Directed I/O, September 2008.

[18] R. Jones. Netperf: A network performance benchmark, version 2.45, 2009. `http://www.netperf.org`.

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4):263–297, November 2000.

[21] B. Leslie, P. Chubb, N. Fitzroy-dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20, 2005.

[22] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[23] A. Menon, S. Schubert, and W. Zwaenepoel. Twin-Drivers: semi-automatic derivation of fast and safe hypervisor network drivers from guest os drivers. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, Washington, DC, March 2009.

[24] PCI-SIG. *PCI local bus specification*, revision 3.0 edition, February 2004.

[25] PCI-SIG. *PCI Express 2.0 base specification*, revision 0.9 edition, September 2006.

[26] M. J. Renzelmann and M. M. Swift. Decaf: Moving device drivers to a modern language. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[27] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proceedings of the ACM EuroSys Conference*, Nuremberg, Germany, March 2009.

[28] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, October 2009.

[29] M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), November 2006.

[30] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 22(4), November 2004.

[31] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 1997.

[32] VMware. Configuration examples and troubleshooting for VMDirectPath. `http://www.vmware.com/pdf/vsp_4_vmdirectpath_host.pdf`.

[33] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[34] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[35] E. Witchel, J. Rhee, and K. Asanovic. Mondrix: memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[36] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 225–240, San Diego, CA, December 2008.

Proxychain: Developing a Robust and Efficient Authentication Infrastructure for Carrier-Scale VoIP Networks

Italo Dacosta and Patrick Traynor
Converging Infrastructure Security (CISEC) Laboratory
Georgia Tech Information Security Center (GTISC)
Georgia Institute of Technology
$\{$*idacosta, traynor*$\}$*@cc.gatech.edu*

Abstract

Authentication is an important mechanism for the reliable operation of any Voice over IP (VoIP) infrastructure. Digest authentication has become the most widely adopted VoIP authentication protocol due to its simple properties. However, even this lightweight protocol can have a significant impact on the performance and scalability of a VoIP infrastructure. In this paper, we present *Proxychain* – a novel VoIP authentication protocol based on a modified hash chain construction. Proxychain not only improves performance and scalability, but also offers additional security properties such as mutual authentication. Through experimental analysis we demonstrate an improvement of greater than 1700% of the maximum call throughput possible with Digest authentication in the same architecture. We show that the more efficient authentication mechanisms of Proxychain can be used to improve the overall security of a carrier-scale VoIP network.

1 Introduction

Voice over IP (VoIP) is fundamentally reshaping the telephony landscape. Instead of using dedicated, circuit-switched lines, VoIP allows for phone calls to be multiplexed with other data traffic over the Internet. This convergence between voice and data communications provides a number of benefits. For instance, providers can now offer a range of new services such as video and presence. Unfortunately, the transition from traditional phone networks to VoIP also creates a number of new security challenges.

Authentication represents one of the most important security issues facing VoIP systems. Providers have responded by implementing a number of security mechanisms, ranging from SSL/TLS to Digest authentication. Unfortunately, none of the suggested schemes are simultaneously strong, efficient *and* scalable enough to meet the needs of carrier-scale networks without vastly increasing the amount of deployed infrastructure.

In this paper, we develop *Proxychain*, a robust and efficient authentication infrastructure designed to support operations in carrier-scale VoIP networks. Our solution is built around a single centralized authentication service working with proxy nodes distributed across a wide geographic area. We reduce the impact of the latency and load associated with this architecture by using a modified hash chain construction (a sequence of one-time authentication tokens generated by applying a hash function repeatedly, once-per token, to a secret root value). In addition to providing an efficient mechanism for mutual authentication, our approach also provides improved scalability through the secure caching of temporary authentication tokens at the proxies. To the best of our knowledge, Proxychain is the first protocol that applies the idea of hash chains in the SIP domain. Proxychain not only adapts this idea to SIP authentication but also extends it by including additional modifications that solve some of the weaknesses associated with hash chain protocols, resulting in a more robust protocol.

This paper makes the following contributions:

- **Design and implementation of Proxychain:** We develop a construction based on modified hash chains. Our construction not only dramatically reduces the load on the centralized authentication database and the latencies associated with accessing it, but also provides mutual authentication for clients and providers.
- **Evaluation of Proxychain through an extensive measurement study:** We measure, characterize and compare the performance characteristics of our proposed infrastructure against commonly used mechanisms. Our results show up to a 1700% improvement over such schemes. Moreover, we demonstrate the ability to support the authentication needs of a national-scale VoIP network using unoptimized COTS hardware and databases.
- **Evidence of robustness to outages and downtime:**

We demonstrate that our construction allows the network to operate during planned and unplanned outages, and estimate its robustness to such incidents. We show the ability to support normal operations with high availability for approximately 6 hours using only 28 minutes of preemptive computation.

Improvements to the efficiency of SIP authentication afforded by Proxychain allow us to significantly increase the overall security of VoIP systems. For instance, several recently disclosed attacks on VoIP systems [2, 29] can be mitigated by simply having an authentication infrastructure scalable enough to cryptographically verify the origin of multiple SIP signaling request types (e.g., INVITE and BYE).

The remainder of our paper is organized as follows: Section 2 discusses a nation-wide VoIP architecture and provides important background information; Section 3 details our proposed protocol; Section 4 provides the details of our experimental setup; Section 5 shows the results of our experiments; Section 6 discusses a number of additional points; Section 7 presents related work; Section 8 offers concluding remarks and future work.

2 A Nationwide VoIP Infrastructure

Telephony networks have long relied on a series of distributed databases and proxies to implement authentication. However, advances in processor speeds and ease of management have prompted a number of cellular [18] and VoIP providers such as Skype to rely on a central authentication service.[1] Therefore, our authentication mechanism is designed to work for a similar architecture. Figure 1 shows our simulated testbed.

2.1 Session Initiation Protocol

The Session Initiation Protocol (SIP) [21] is the underlying architecture of the majority of VoIP systems. This application-layer signaling protocol has several components. End devices are known as *User Agents* (UA), and can act as a client (UAC) or as a server (UAS). UACs generate SIP requests, while the UAS generates responses to SIP requests. When attempting to establish a session with Bob, Alice sends her request to a SIP *Proxy* server. The proxy determines the IP address of Bob's UAS and forwards Alice's request. In addition to routing call setup requests, a proxy also participates in the process of authentication with the help of an authentication database.

SIP provides two message types: requests (client to server) and responses (server to client). There are six types of requests: *INVITE* (to establish a session between UAs), *ACK* (to acknowledge a reliable message exchange), *CANCEL* (to terminate a pending request), *BYE* (to terminate a existent session), *OPTIONS* (to query for

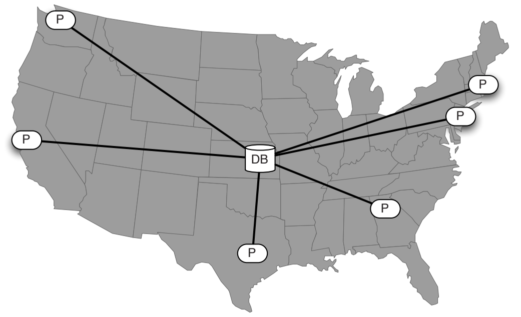


Figure 1: The hypothetical nationwide SIP infrastructure modeled in our experiments using latencies collected from Planetlab. As it is done by some cellular providers, our authentication service (DB) is centrally located, with proxies (P) distributed across the country.

the capabilities of servers), and *REGISTER* (used by a UA to notify its current IP address to a Registrar process running on the proxy). The responses are grouped into six categories and indicate the status of a current request. For example, a *200 OK* response indicates a successful transaction (more detail in the RFC 3261 [21]).

2.2 Digest Authentication

SIP Digest authentication is a challenge-response authentication protocol based on HTTP Digest authentication [11]. Digest authentication is used by SIP proxies to validate the identity of requests received from UAs. It allows users to prove their knowledge of a shared secret (e.g., password) to a server without sending the secret unprotected over the network (protection against eavesdropping attacks). Digest authentication is widely supported because it is more efficient and easier to implement than the other protocols recommended by RFC 3261 (i.e., TLS, S/MIME, IPsec). Furthermore, it is the only authentication protocol required in the UAs according to RFC 3261 (support for other protocols is not required).

Figure 2 shows a SIP call dialog using Digest authentication. As in most deployments, only INVITE requests require authentication. First, Alice's UA sends an INVITE request to the proxy. The proxy determines that the request requires authentication and responds with a SIP 407 response ("`Proxy Authentication Required`") containing a nonce. Alice's UA acknowledges the reception of the challenge, computes the hash of the shared secret and the nonce and sends it back to the proxy using a new INVITE message. The proxy then computes the answer after querying a database that stores the user's shared secret. Finally, the proxy compares both values and, if they match, forwards the INVITE to the destination and the SIP dialog continues its standard flow.

Digest authentication efficiency relies on the use of hash operations and nonces, instead of symmetric or public key cryptography. In its basic form, a Digest authentication response is computed as follows:

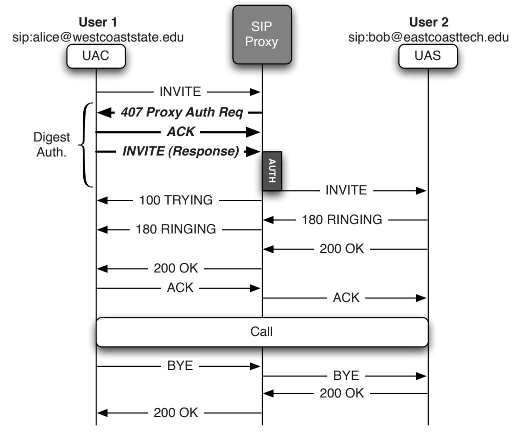


Figure 2: SIP call setup using Digest authentication (bold).

Response = *H*(*H*(*uid*||*realm*||*pwd*) ||*n*|| *H*(*method*||*URI*))

where $H()$ is a cryptographic hash function (MD5 is the default), || corresponds to a concatenation operation, *uid* is the user ID, *realm* is the proxy's protection domain, *pwd* is the user password, n is a nonce, *method* is the SIP request authenticated (e.g., INVITE) and *URI* is the destination address Alice is trying to reach (e.g., bob@eastcoaststate.edu).

2.3 Problems with Digest Authentication

While more efficient, Digest authentication is less secure than protocols such as TLS or IPsec. For instance, it does not provide mutual authentication and complete message integrity. Limited integrity protection is offered but it is optional and not widely supported by UAs. Additionally, current implementations actually send the shared secret from the database to the proxy in order to calculate the correct client response. This approach is dangerous if the proxy is compromised. Several vulnerabilities have been published regarding commercial SIP deployments due to these weaknesses [29].

The use of Digest authentication in an environment with a remote authentication service dramatically reduces performance. The main reason is that authentication operations become more expensive - the round-trip time (RTT) between a proxy and the database (tens of milliseconds) is now added to each authentication operation (hundreds of microseconds). The additional time added per call setup reduces the call throughput of each proxy. The problem is exacerbated by the fact that proxies have to query the database for each SIP message that requires authentication. This action also creates a considerable network load when the call throughput is high. If multiple proxies are used, the load could overwhelm the database or its network link. As a result, scalability is also affected.

The use of multiple databases (i.e., one local database per proxy) or adding more hardware resources to the database are not efficient solutions. Dacosta et al. [8] showed that the effects of network latency could be reduced by a combination of parallelization and batching techniques. However, the network load to the database is still high enough to affect the scalability of the system. A more efficient approach is to reduce the number of queries to the database. To achieve this, we can use temporary authentication credentials that each proxy stores in memory and that can be used in multiple authentication operations without contacting the database. This approach reduces the load received by the database and the effects of network latency. Our proposed protocol follows this approach.

3 Proxychain Protocol Specification

3.1 Hash Chains

A hash chain is created by applying a cryptographic hash function $H()$ (e.g., MD5, SHA-1) multiple times to a random value r to generate a sequence of values that can be used as one-time authentication tokens. A hash chain of length n is computed as:

$$H^n(r) = H(\dots H(H(r))\dots)$$

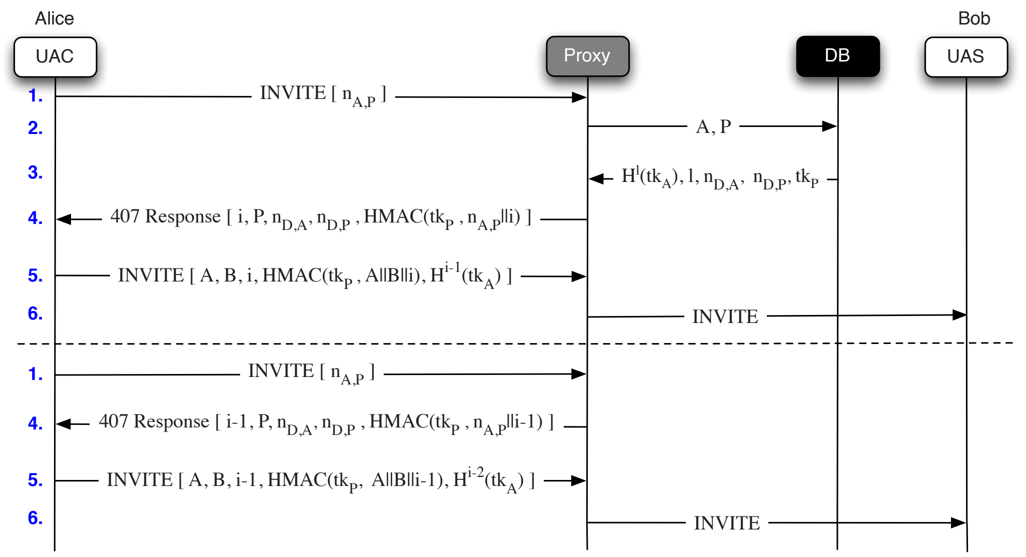
Hash chains rely on the preimage resistant (i.e., one-way) property of cryptographic hash functions. When attempting to authenticate to a server possessing $H^n(r)$, the client transmits $H^{n-1}(r)$. The server then hashes $H^{n-1}(r)$ a single time and, if the result matches $H^n(r)$, authenticates C based on the computational infeasibility of an adversary guessing the correct preimage.

3.2 Design Goals

Proxychain design addresses some of the shortcomings of Digest authentication in SIP topologies with a centralized authentication service. Our first goal is *efficiency:* Proxychain should execute authentication operations faster than Digest authentication, allowing improved call throughput. Second, we focus on *scalability:* Proxychain should support more users and proxies than Digest authentication without the need for additional resources. In particular, Proxychain should reduce the bandwidth and processing time required by the database to avoid bottlenecks. Finally, our third goal is *security:* Proxychain should improve upon the security assurances provided by Digest authentication.

3.3 Design and Formal Description

Proxychain is designed to reduce the impact of latency and load on the remote authentication service by caching



Alice
UAC
Proxy
DB
Bob
UAS
1. INVITE [n_{A,P}]
2. A, P
3. H^l(tk_A), l, n_{D,A}, n_{D,P}, tk_P
4. 407 Response [i, P, n_{D,A}, n_{D,P} , HMAC(tk_P , n_{A,P}||i)]
5. INVITE [A, B, i, HMAC(tk_P , A||B||i), H^{i-1}(tk_A)]
6. INVITE
1. INVITE [n_{A,P}]
4. 407 Response [i-1, P, n_{D,A}, n_{D,P} , HMAC(tk_P , n_{A,P}||i-1)]
5. INVITE [A, B, i-1, HMAC(tk_P, A||B||i-1), H^{i-2}(tk_A)]
6. INVITE

Figure 3: Call setup flow using Proxychain. For the first request (above dashed line), the proxy must request a temporary credential from the database. Subsequent requests (below dotted line) can be dealt with immediately by the proxy.

1. $A \rightarrow P : A, B, n_{A,P}$
2. $P \rightarrow D : A, P$
3. $D \rightarrow P : H^l(tk_A), l, n_{D,A}, n_{D,P}, tk_P$
4. $P \rightarrow A : i, P, n_{D,A}, n_{D,P},$ $HMAC(tk_P, n_{A,P}||i)$
5. $A \rightarrow P : A, B, i, HMAC(tk_P, A||B||i),$ $H^{i-1}(tk_A)$
6. $P \rightarrow B$

A, B, P, D	:	Alice, Bob, Proxy, Database
$k_{A,D}$	:	Secret key between Alice and database
$n_{D,A}, n_{D,P}, n_{A,P}$	:	Nonces
l	:	Hash chain length
i	:	Hash chain current sequence number
$H^i(x)$	:	*i-th* hash value of *x*, H(H(...H(*x*)...))
HMAC(k, x)	:	HMAC with key *k* on *x*
tk_A	:	HMAC$(k_{A,D}, n_{D,A}\|\|P)$
tk_P	:	HMAC$(k_{A,D}, n_{D,P}\|\|P)$

Figure 4: *Proxychain protocol:* The formal definition of the Proxychain protocol. We assume that there exists an encrypted channel (e.g., IPsec connection) between the proxy and the database.

temporary authentication credentials at the proxies. Using hash chain-based credentials of length l, a proxy can authenticate multiple requests from a particular user with only $\frac{1}{l}$ queries to the database. The database creates credentials based on the secret it shares with each user and determines the credential's parameters, including length, hash function, and expiration time. This approach is more secure than the associated Digest authentication mechanism, as the shared secret between the database and the user is never exposed to the proxies. A compromise of one of these servers, therefore, does not necessarily require password resets for large number of users.

Each proxy provides services only to users that are geographically close to it (i.e., based on IP address or ZIP code information), much like a traditional telephony switch. Each proxy accordingly needs to store credentials for only a subset of the total number of users in the system. We explore the overhead associated with such credential storage in Section 4.2.

Figures 3 and 4 provide graphical and formal definitions of the Proxychain protocol, respectively. A user Alice attempts to call Bob by first sending an INVITE request to her proxy, which contains the source and destination of the call and a nonce $n_{A,P}$ (Message 1). The proxy checks to see if it has a credential for Alice and, if not, queries the authentication database with the identifiers corresponding to Alice and the proxy (A, P) for a new hash chain (Message 2). Note that requests between proxies and the authentication database occur over a long-lived, encrypted and authenticated channel such as IPsec

or TLS/SSL. The database generates a five-tuple that includes a new hash chain $(H^l(tk_A))$, the length of the hash chain l, nonces for both the proxy and Alice ($n_{D,P}$ and $n_{D,A}$), and a session key tk_P. The hash chain is calculated as H^l(HMAC$(k_{A,D}, n_{D,A}||P)$), and the session key as HMAC$(k_{A,D}, n_{D,P}||P)$ (Message 3).

After receiving the tuple from the authentication database, the proxy returns a `407 Proxy Authentication Required` SIP message to Alice. This message includes a counter $i \leq l - 1$, the proxy's identifier P, the two nonces generated by the authentication database ($n_{D,P}$ and $n_{D,A}$) and a network authentication token HMAC$(tk_p, n_{A,P}||i)$ (Message 4). The client receives the response and uses $k_{A,D}$ to calculate the session key tk_P and then authenticates the message from the proxy. If the message authenticates properly, Alice then generates her session key tk_A and hashes it $i - 1$ times to generate $H^{i-1}(tk_A)$. Alice responds to the proxy by sending a new INVITE message containing A, B, i, $HMAC(tk_P, A||B||i)$ and $H^{i-1}(tk_A)$, which the proxy hashes forward a single time (assuming that the HMAC properly verifies) (Message 5). If $H(H^{i-1}(tk_A)) = H^i(tk_A)$, then the proxy records $H^{i-1}(tk_A)$ as the next legitimate credential, decrements i and the INVITE request is forwarded to Bob (message 6). On subsequent authentication attempts by Alice where $c < i - 1$, the proxy responds to Message 1 with Message 4, which contains $c, P, n_{D,A}, n_{D,P}, HMAC(tk_P, n_{A,P}||c)$.

Note that unlike Digest authentication, Proxychain provides mutual authentication. Specifically, the network authentication token $HMAC(tk_p, n_{A,P}||i)$ can only be produced with knowledge of tk_P and using the nonce supplied by the user Alice. Moreover, because only the user and the authentication database could have created tk_P (because only they have knowledge of $k_{A,D}$), an adversary can not create legitimate hash chains without the assistance of the authentication database.

4 Experimental Setup

4.1 Experimental Testbed

Our experimental testbed is based on the VoIP infrastructure depicted in Figure 1. As this figure shows, the testbed is composed of three main components: the authentication database, SIP proxies and the user clients (UAs). The database and proxies are run on servers from the Georgia Tech Emulab testbed.[2] We use seven servers to represent the infrastructure (one database and six proxies). These servers run Linux Kernel 2.6.26 (Fedora Release 8), have two 2.80 GHz Intel Xeon processors and 512 MB of memory. The UAs are run on servers from our research lab. A total of nine servers are used, each running multiple UA instances to generate call traffic. These servers run Linux Kernel 2.6.24 (Ubuntu 8.04.2), eight (8) 2.00 GHz Quad-Core AMD Opteron processors and 16 GB of memory.

The network latency between the proxies and the database is simulated using Emulab's traffic shaping functionality. In order to use realistic latency values, we performed measurements using the Planetlab network testbed.[3] Using the ping network tool, we measured the round-trip time (RTT) between a Planetlab node located in the University of Kansas and Planetlab nodes located at UC Berkeley (67.6 ms), Georgia Tech (33.1 ms), MIT (44.7 ms), Princeton (43.8 ms), the University of Texas (20.6 ms), and the University of Washington (43.4 ms). The RTT data was collected during a 24 hours period and average values were calculated. Finally, no additional latency values were simulated between the proxies and the UAs (latency was around 1 ms). The reason is that our testbed assumes physical proximity and low latency values (e.g., $<$ 10 ms) between the UAs and the proxies. Simulating this latency is not necessary because it would not affect the test load generated by the UAs and our results (it would slightly affect the setup time of each call).

The proxies are implemented using OpenSIPS[4] 1.5.2. OpenSIPS is a mature open source SIP proxy optimized for high performance. The proxies are configured with minimal functionality (stateless configuration and basic modules required for routing). We run MySQL[5] 5.0.45 as our database, a well-known open source relational database management system. MySQL is run with a default configuration (no optimizations). Finally, SIPp[6] 3.1 is used to generate the UAs' workload, which conforms to a uniform random distribution. SIPp is an open source traffic generator for the SIP protocol. A total of 36 SIPp instances are used in our experiments (18 UACs and UASs). Default SIPp scenarios are modified to support INVITE and BYE authentication for Digest and Proxychain authentication (SIP call flows in Figures 2 and 3).

Each proxy serves requests for 200,000 unique users. The number of users per proxy is limited by the proxy's available memory, disk space in the database and the size of authentication credentials (see Section 4.2). As a result, the total number of users in the database is 1,200,000. All the users are part of a single SIP domain (no inter-domain calls).

4.2 Proxychain Implementation

Implementing Proxychain requires a combination of new code modules and modifications to existing software. In the proxies, OpenSIPS ($\approx$ 320000 lines of code (loc)) required approximately 710 loc to support Proxychain. In the UAs, SIPp ($\approx$ 3000 loc) required around 140 loc. In the database, we built a separate concurrent-process server application to handle queries from proxies and the associated cryptographic operations. This server application required approximately 880 loc. The

MySQL database software itself was unmodified. All of our experimental code, which was written in C, and supporting scripts are available at `http://www.cc.gatech.edu/~idacosta/proxychain.html`

Proxychain uses the same SIP headers in the challenge and response messages. For example, a Proxy-Authenticate header (challenge) looks as follows:

```
Proxy-Authenticate:PC realm="CISEC", i="10",
nda="0ec497d9a5ba5e1f2b2177d83fb3d341",
ndp="f1e992583dd5daecddea3309a01e5347",
hmac="15f5d33206e79eaea7245682d9953164"
```

where *PC* indicates the use of the Proxychain protocol, *realm* is proxy's identifier, *i* is the sequence number, *nda* and *ndp* are the nonces and *hmac* is the network authentication token.

The corresponding Proxy-Authorization header using Proxychain looks as follows:

```
Proxy-Authorization:  PC username="0000001",
realm="CISEC", i="10",
response="a0843d4b8a712284ff5a6fcd136c4b47,"
hmac="f9fd4ef6689850406a560965a4381c57"
```

where *response* is the next value in the hash chain sequence. The other parameters have the same meaning as in the Proxy-Authenticate header.

Our Proxychain implementation uses the MD5 hash function in order to compare it more directly and fairly to Digest authentication. Nevertheless, our code requires few modifications to support SHA-1. With MD5, the size of a temporary authentication credential is 134 bytes. As a result, each proxy in our testbed requires a minimum of 26 MB of free memory for serving 200,000 users.

4.3 Methodology

We perform a number of different experiments in order to characterize Proxychain. We specifically compare our protocol against a system with no authentication mechanism and one using Digest authentication. We do not measure more computationally expensive mechanisms such as TLS/SSL as previous studies have demonstrated that they provide significantly lower throughput [5, 6, 15, 16]. We collect the following metrics in most of our experiments: call throughput, message retransmissions, failed calls, bandwidth utilization and database CPU utilization. These are global metrics, the totals for the whole infrastructure (i.e., the call throughput is equal to the sum of the call throughput measured in each proxy).

The *call throughput* refers to the number of successful calls per second (cps) measured every five seconds.

Message retransmissions corresponds to the number of SIP messages retransmitted due to the expiration of timers in the UAs. Our tests use the default retransmission time

Protocol	Digest	Stdev	Proxychain	Stdev
Response (μsec)	116.81	13.59	184.76	49.92
Verification (μsec)	197.24	21.51	66.97	15.07

Table 1: Response computation time at the UA and verification time at the proxy for Digest and Proxychain authentication. Proxychain adds little overhead to the response computation and it is more efficient performing verifications.

Length	10	100	1000	10000
Time (μsec)	294.10	335.15	1383.53	11875.71
Stdev (μsec)	18.42	15.28	18.07	120.44

Table 2: Time required by the database to compute credentials with different hash chain lengths. For lengths < 100, the overhead is small.

defined by SIP standards (500 ms). *Failed calls* refer to the total number of unsuccessful calls measured in the last period. In our experiments, we consider only calls failures due to maximum number of retransmissions (maximum number of UDP retransmissions attempts has been reached). We use the default values in SIPp for the maximum number of retransmissions: five for INVITE messages and seven for others. Finally, *bandwidth utilization* corresponds to the total network throughput (KBytes/sec) measured from the database during each test.

During our experimental analysis, each test was run at least 10 times to ensure the soundness of the results. Average values are used in our analysis and a 95% confidence interval is provided in most of the graphs. Note that these bounds are often difficult to observe in our graphs as the values are generally very close to the mean.

5 Experimental Results

5.1 Microbenchmarks

To understand the computational differences between Digest and Proxychain authentication, we measure the time to compute a response in the UA and the time to verify a response in the proxy. To measure these values, we use network traces (100 samples per value). For Proxychain, the measurements are performed the first time a credential is used (hash chain length of 10). This corresponds to the worst case for response computation because it requires the highest number of hash operations (9 operations).

Table 1 shows the results. The UA running Proxychain requires approximately 70 μsec of additional computation than one running Digest authentication. This difference is due to the additional integrity checks and hash operations required by Proxychain in the UA. However, this difference is not significant as individual UAs does not perform large amounts of computation in this system. Interestingly, the response verification is nearly three times faster when Proxychain is used by the proxy. The rea-

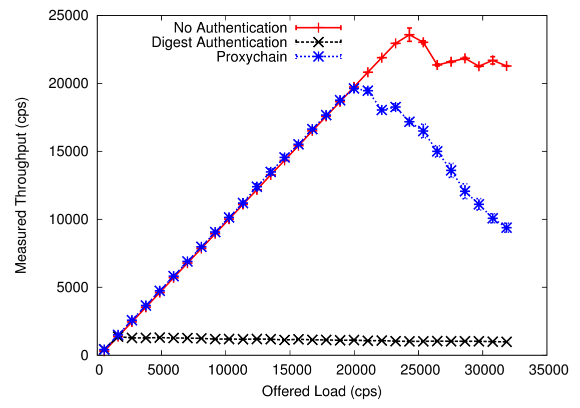


Figure 5: Total call throughput for no, Digest and Proxychain authentication. Proxychain's maximum call throughput is close to the one obtained without authentication.

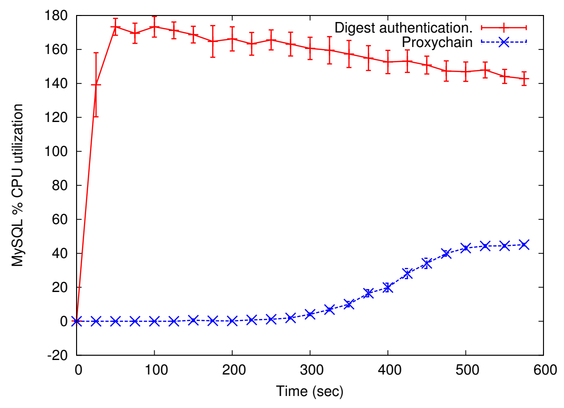


Figure 6: Percentage of CPU required by the database process for Digest and Proxychain authentication. The database process is virtually idle when Proxychain is used.

son is that Proxychain only requires two hash operations to verify a response. On the contrary, Digest authentication requires three hash operations and additional checks to verify a response. Based on these results, we argue that the computational overhead added by Proxychain is not significantly different from the one added by Digest authentication.

We also evaluate the overhead of generating hash chains of varying lengths. Specifically, we measure the time required by the authentication database to generate credentials of lengths 10, 100, 1000 and 10000. As before, we use network traces to measure the time for each configuration (100 samples per configuration). Table 2 shows the results of these experiments. As expected, increasing the hash chain size increases the time required to generate credentials. The additional time remains small for hash chains with length up to 100 (< 350 μsec).

5.2 Call Throughput

Microbenchmarks provide insight into the overhead that can be expected at each component of the network. However, they do not provide a picture of the overall behavior of a system. Accordingly, we characterize the interaction of those components by measuring total call throughput. We compare throughput for systems configured to use Digest authentication, Proxychain and no authentication mechanism. UAs generate an increasing call load (270 cps increments every 5 seconds) over the course of 10 minutes. In addition, we evaluate the best configuration for each protocol. For Digest authentication, we use close to 100 concurrent proxy-processes per proxy.[7] For Proxychain, we preload each proxy with all its user credentials (200K credentials with hash chain length of 10) before each experiment and use 8 concurrent proxy-processes per proxy (OpenSIPS recommended value).

Figure 5 shows the results of these experiments. Without authentication (baseline configuration), the network supports a maximum call throughput of nearly 24,000 cps. When Digest authentication is used, the maximum call throughput drops dramatically to approximately 1,160 cps. This result represents a 95% reduction in call throughput when compared with the baseline configuration. For Proxychain, the result is more favorable: a total call throughput of over 19,700 cps. In this case, the call throughput drops by only 18% when compared with the baseline configuration. However, when compared to Digest authentication, Proxychain allows an increase of over 1,700% (more than an order of magnitude). Accordingly, Proxychain is significantly more efficient than Digest authentication in this architecture.

Figure 6 provides insight into the poor performance of Digest authentication. The database process rapidly reaches 175% CPU utilization (dual-core machine). This behavior indicates that queries from the proxies saturate the authentication database, making it a bottleneck. We observe the opposite when using Proxychain. The database was virtually idle ($< 5\%$ CPU utilization) before the system reaches its maximum call throughput, at which point the system becomes unstable due to the high number of retransmissions.

A naïve solution to improve Digest authentication performance would be to use a more powerful database. Therefore, we repeated the experiment using a quad-processor server for the database. As expected, the maximum call throughput increases, but only to approximately 4,000 cps. However, in this experiment the database does not saturate - CPU utilization is below 300%. In this case, throughput fails to increase further due to the network latency between the proxies and the database.

Another important difference is the total bandwidth required for both configurations. The message overhead between a UA and the proxy are arguably equivalent. Message 4, the challenge, requires an additional 92 B and 165 B for Digest and Proxychain authentication, respectively. The response in Message 5 similarly requires an addi-

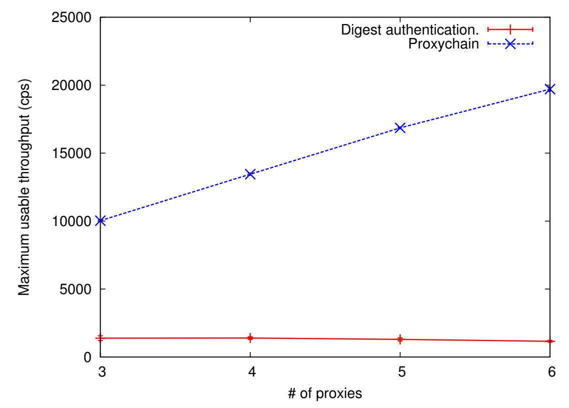


Figure 7: Throughput measured for a range of proxies using Digest and Proxychain authentication. Proxychain is considerably more scalable than Digest authentication.

tional 199 B and 153 B. At its maximum call throughput (measured from the database), Digest authentication required almost 130 and 430 KBytes/sec for queries and responses respectively. In contrast, Proxychain required less than 1 KByte/sec for both, queries and responses. As expected, the use of temporary credentials significantly reduces the total number of queries to the database.

The previous results also mean that increasing the hash chain length ($>$10) will not help to improve performance in our testbed. The reason is that the load in the database is already low with a hash chain length of 10. Using a longer size will make the load even lower but the difference will not affect the overall performance of the system. On the contrary, using hash chains that are too long could affect performance because of the additional hash operations that will be needed by the UACs and the database.

Finally, for the baseline and Proxychain configurations, the maximum call throughput is limited by the proxy application itself: OpenSIPS. Analyzing the resources usage statistics (memory, CPU and bandwidth) collected during the experiments for the different testbed components, we find that none of the resources are completely used (no shortage of resources) when the two configurations reach the maximum call throughput. Based on this evidence and in our experience with OpenSIPS, we can conclude that the OpenSIPS software is the performance bottleneck for no authentication and Proxychain configurations. Using an optimized version of OpenSIPS or a faster proxy server application will provide higher call throughput values.

5.3 Scalability

In this set of experiments, we evaluate how the testbed handles an increasing number of users, and therefore, an increasing load. To simulate a varying number of users, we measure performance with a varying number of proxies, where each proxy represents 200,000 users. Using a similar procedure as in the previous test, we measure the call throughput for 3, 4, 5 and 6 proxy configurations (600K, 800K, 1M and 1.2M users respectively).

The results are presented in Figure 7. We can see that for Digest authentication, the maximum call throughput measured is approximately the same ($\approx$1,200 cps; linear regression: $y = -79.6x + 1670.5$ $R^2 = 0.848$ [8]) for all the configurations. The reason is that even for a three-proxy configuration, the database becomes saturated rapidly (see previous test). Therefore, Digest authentication limits the scalability of the system. For Proxychain, the maximum call throughput increases linearly with the number of proxies ($\approx$3,250 cps per proxy; linear regression: $y = 3243.9x + 416.5$ $R^2 = 0.998$). From these results, we can conclude that Proxychain allows the system to grow by just adding new proxies and without requiring changes to the database.

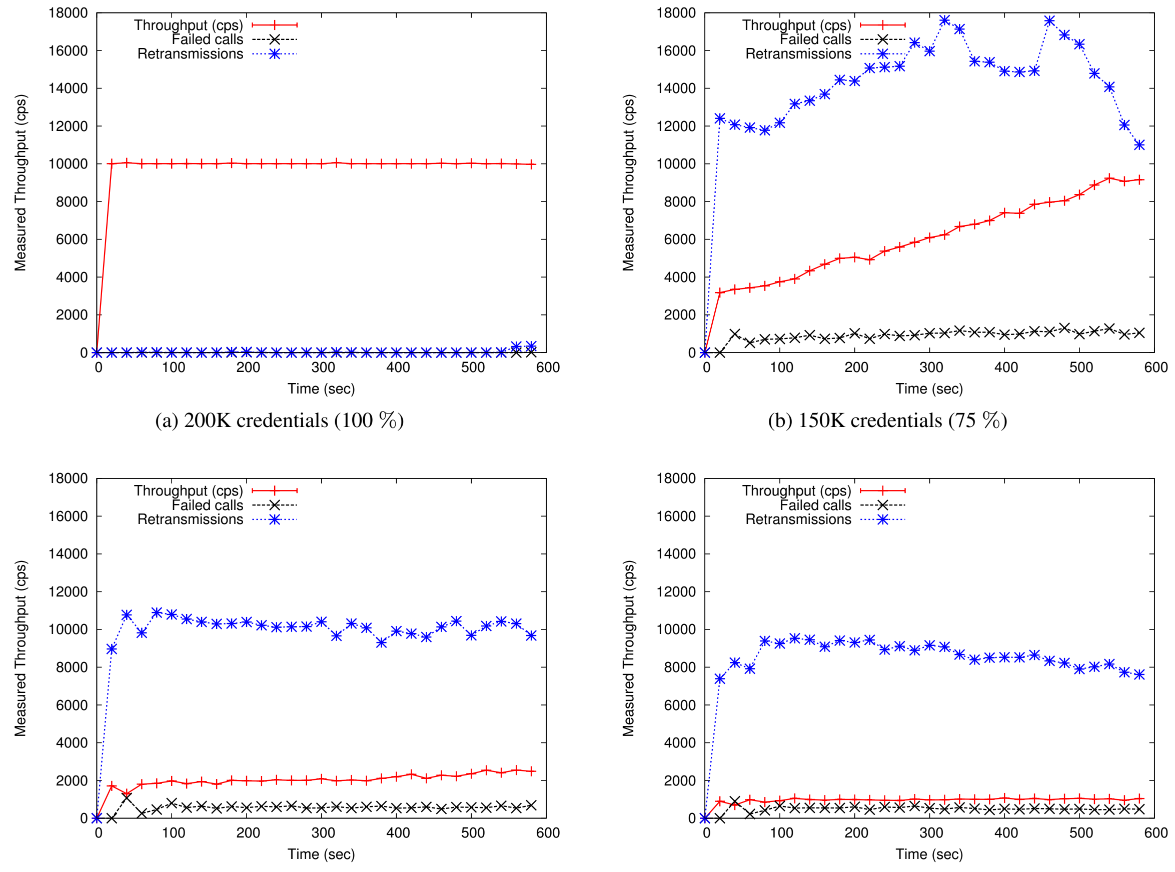
5.4 Credential Preloading in the Proxies

In the previous tests, we evaluated Proxychain's performance using a best-case scenario: each proxy had all the credentials in memory before the tests started. We now evaluate performance when a lower number of credentials are preloaded in each proxy. For this purpose, we use a similar procedure as in previous tests but with two exceptions. First, we use a constant workload of 10,000 cps with no ramp-up period. Second, we preload the proxies with 200K, 150K, 100K and 50K credentials in each test.

Figure 8 shows the results for all the configurations. For the 200K configuration (best-case, Figure 8a), the call throughput reaches 10,000 cps quickly ($<$ 10 sec) with virtually no message retransmissions or failed calls. For the 150K configuration (Figure 8b), the call throughput jumps to approximately 3,000 cps, and then continues increasing until it reaches almost 10,000 cps by the end of the test. However, a large number of retransmissions and failed calls occur. Finally, for the other two configurations (Figures 8c and 8d), the behavior is worse. The maximum call throughput measured was around 2,000 and 1,000 cps respectively during the experiments. The number of retransmissions and failed calls is also constantly high. In theory, each configuration should have reached 10,000 cps after some period of time. However, the large number of retransmissions makes the system unstable. These results show the importance of having most of the credentials stored in the proxies to avoid the negative effects of retransmissions, especially when high loads are expected.

5.5 Prefetching mechanism

The previous test shows that Proxychain is more effective if each proxy has credentials for almost all its users (best case scenario). However, credentials are stored or updated in the proxy only after a user request that requires authentication. Therefore, we implement a prefetching mechanism that automatically queries the database for credentials without requiring any user action. This mechanism, running as a separate proxy process, checks if a user has a credential in the proxy or if her credential has already expired (i.e., l = 0). In short, the prefetching mechanism guarantees the best case scenario for Proxychain.



(a) 200K credentials (100 %)

(b) 150K credentials (75 %)

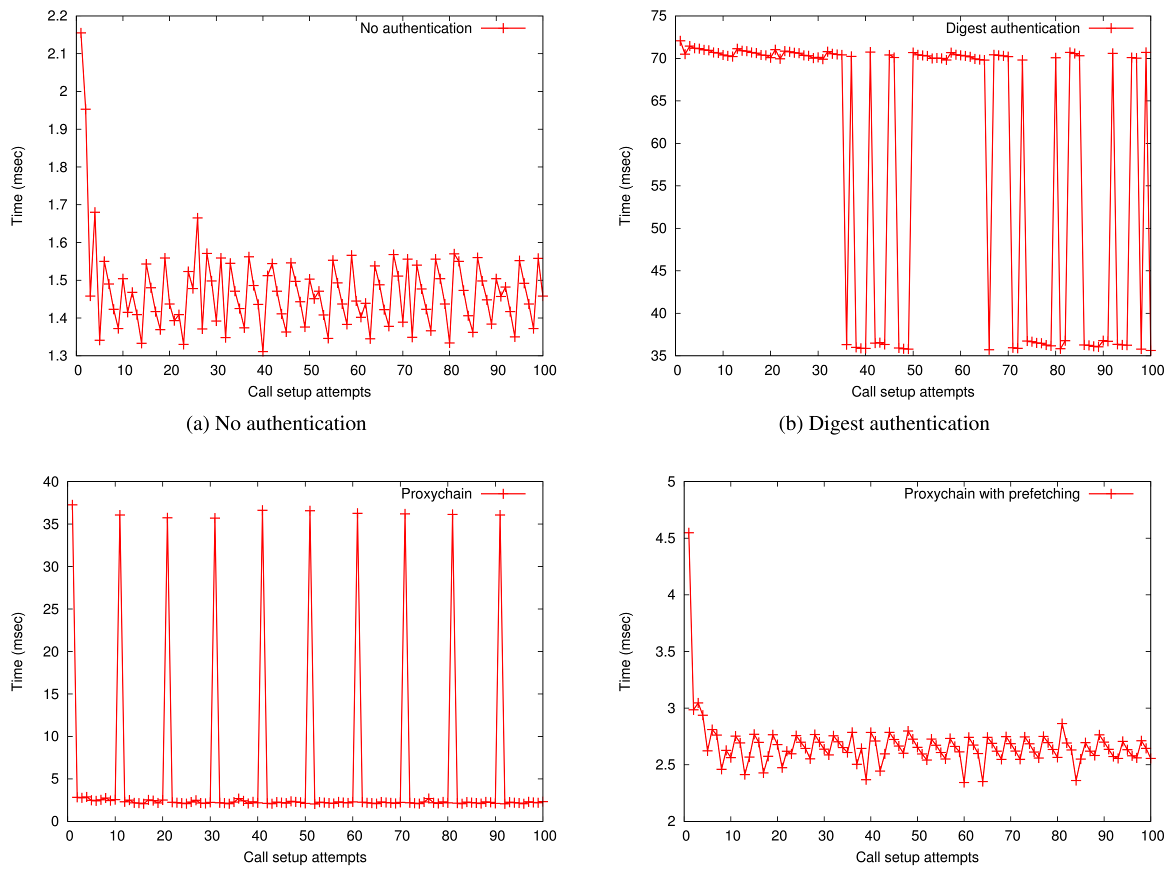
(c) 100K credentials (50 %)

(d) 50K credentials (25 %)

Figure 8: Call throughput measured for different number of credentials preloaded in the proxies and a constant offered load (10K cps). Proxychain requires that proxies have most of the credentials in memory for maximum performance.

In this experiment, we characterize the effect of the prefetching mechanism on the call setup time for individual UAs (time elapsed between the first INVITE request and the 200 OK response). We use a UA sending a low load ($<$ 5 cps) to a single proxy and estimate the call setup time using network traces (100 samples). Four proxy configurations are used: no authentication, Digest authentication, Proxychain and Proxychain with prefetching.

Figure 9 shows the results for each configuration. As expected, when no authentication is used (Figure 9a), the call setup is the fastest: 1.47 ms on average. For Digest authentication (Figure 9b), we can observe the effects of the RTT between the proxy and the database ($\approx$33 ms) on the call setup time. Two call setup times are measured: 36 and 71 ms approximately. The reason is that for the first value, only one RTT is required during call setup, while for the second value, two RTTs are required due to the low test load used (no TCP piggybacking). In general, only one RTT is required, so we can assume that the call setup time for Digest authentication is approximately 36 ms. In the case of Proxychain, Figure 9c shows how the temporary credentials reduced the call setup time while they are valid. While the credentials are active (hash chain size $>$ 0), the call setup time is only 2.27 ms on average. Once a credential expires (hash chain size = 0), a query to the database is required, so the call setup time increased by one RTT: 36.28 ms on average. When Proxychain is used with prefetching (Figure 9d), the average call setup time is only 2.67 ms. The reason is that no credential updates are performed during call setups. Instead, credentials are updated by the prefetching process automatically, before they are required in a call setup. Therefore, the call setup time when Proxychain is used with prefetching is close to the call setup time when no authentication is used ($\approx$1 ms difference). Accordingly, prefetching helps to eliminate the effect of network latency on call setup time.



(a) No authentication

(b) Digest authentication

(c) Proxychain

(d) Proxychain with prefetching

Figure 9: Call setup time for four different configurations: no, Digest, Proxychain and Proxychain with prefetching authentication. The call setup time for Proxychain with prefetching is similar to the one obtained with no authentication.

5.6 Authenticating Multiple Message Types

In our final set of experiments, we explore the effect of authenticating multiple SIP message types request per session (call dialog). For example, the lack of authentication of BYE requests allows several reported attacks against SIP deployments [29]. However, if BYE requests are also authenticated using Digest authentication, the performance of the system will decrease even more due to the additional operations and queries to the database. In this experiment, we evaluate the impact of authenticating INVITE and BYE requests on performance when Proxychain is used. We use a similar procedure as in Section 5.2 (i.e., no prefetching). The only difference is that the proxies and UACs are configured to authenticate BYE in addition to INVITE requests.

Figure 10 shows the call throughput for the two configurations: INVITE and "INVITE and BYE" Proxychain authentication. As expected, the maximum call throughput supported by the testbed decreases when two requests are authenticated to approximately 12,000 cps. This represents a performance drop of nearly 50%. The reason is that credentials are used faster (twice as fast) because two authentication operations are required per call, making the number of queries to the database increase, resulting in higher CPU and bandwidth utilization. However, the use of Proxychain to authenticate two types of signaling messages still provides over 800% greater throughput than Digest authentication authenticating a single message.

Finally, we test if increasing the hash chain length improves the performance in this scenario. The idea is that, if credentials are used faster when two requests per call are authenticated, increasing the hash chain length should reduce how fast they need to be replaced. This will result in lower load to the database and increased throughput. The experiment confirms our hypothesis: using a hash chain length of 20 results in a maximum call throughput of almost 14,000 cps. This represents an improvement of almost 17% when compared to using hash chain length of 10. However, increasing the hash chain length further does not improve performance. On the contrary, the performance drops back to almost 12,000 cps with a hash chain size of 30 (using a longer hash chain caused earlier retransmissions, which affects the performance). The reason for these results is that we again reach the limits of the proxy application. The call throughput achieved is lower because the authentication of two requests involves additional messages and operations.

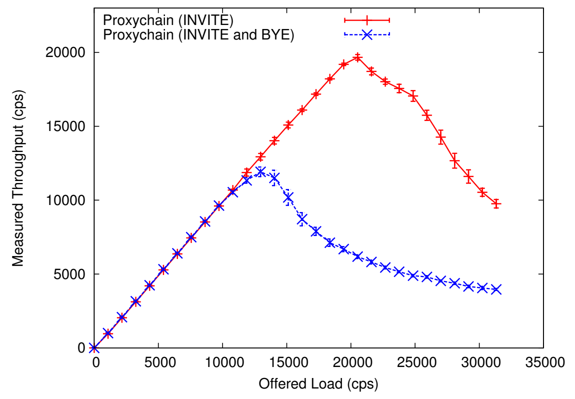


Figure 10: Throughput for INVITE and INVITE and BYE Proxychain authentication. Proxychain allows authentication of two requests per call while still supporting high throughput.

6 Discussion

6.1 Performance

The results presented in the previous section show that Proxychain effectively addresses the limitations of Digest authentication in VoIP topologies with a centralized authentication service. Specifically, Proxychain reduces the effects of network latency, allowing higher throughput. In our testbed, Proxychain's performance improvement was enough to reach the limits imposed by the proxy application (OpenSIPS). Moreover, Proxychain reduces the load received by the database, improving scalability.

The caching of temporary authentication credentials across the proxies allows our solution to perform so much better than Digest authentication. Not surprisingly, cellular networks perform a similar distributed caching of credentials, which are generated by a *Home Location Register* (HLR) and stored in the Mobile Switching Center/Visitor Location Register (MSC/VLR) closest to the client. However, the Proxychain approach is more efficient in terms of memory. Specifically, the current approach used in cellular networks requires that multiple credentials are stored in the MSC/VLR per user. Should the authentication database (HLR) wish to reduce its load, the proxies (MSC/VLRs) would need to be equipped with additional memory. Because Proxychain authentication credentials require a *constant* amount of memory regardless of the hash chain length, our approach is also more scalable than traditional caching. This property is particularly advantageous as it allows for more dynamic behavior by the infrastructure. For example, a database could monitor the received load and automatically increase the length of the hash chains in response to a spike in the load (e.g., busy hours, DoS attack or a flash crowd). We plan to explore such dynamic reprovisioning in future work.

The performance gains obtained in our experiments are based on the assumption that each proxy has most of its users' credentials most of the time. We also assumed that each proxy has a fixed set of registered users and that users do not register with other proxies often (e.g., traveling to another state). These assumptions can be relaxed by providing additional cache space in the proxies. For example, each proxy will have a cache of fixed size, and keep in the cache the credentials of the most active users. When new users register with a proxy, the proxy can use an eviction policy to replace the credentials in the cache based on frequency of use. In this way, each proxy could handle a variable number of users (more flexibility). This approach will be evaluated in future work.

The call throughput numbers achieved in our testbed could be considered high for commercial VoIP deployments. For example, AT&T average nationwide call volume is estimated to be around 300M calls per day, or an average of 3,472 cps [10], or roughly 17% of the throughput provided by our architecture. We note that while our testbed lacks some of the other functionality that a provider may chose to deploy (e.g., billing, media gateways), the performance benefits provided by Proxychain represent a significant potential improvement to real networks. Specifically, the additional capacity offered by Proxychain can serve as a defense mechanism to handle unexpected increments of requests for service.

The performance gains obtained by Proxychain requires some trade-offs. First, a proxy using Proxychain requires to keep a small amount of state for all its users (credentials), which is not necessary for Digest authentication. However, our experiments demonstrated that this was not a significant burden. UACs also need to perform more authentication operations when Proxychain is used. Specifically, Proxychain requires additional integrity checks and hash chain computations required to create a response. Nevertheless, the most expensive operations are hash computations that are in general very efficient to execute. In addition, the use of adequate hash chain lengths (i.e., < 100) and caching intermediate results in the UAC can reduce these overheads. Third, the database also requires to perform computation to create the user credentials. However, this is a one-time cost and it is lower than processing an equivalent number of requests per user as in Digest authentication.

In general, any SIP infrastructure with multiple proxies and a remote central authentication service will benefit from Proxychain, even if the performance requirements are not carrier-level. For example, the SIP infrastructure of a multinational corporation where each regional office has a SIP proxy and the central database is located in the headquarters. The use of Proxychain in this scenario will reduce the load to the database (lower bandwidth and CPU utilization) and provide more security. As our results shows, the main requirement is to cache the credentials of most of the users (e.g., $> 75\%$) served by each proxy. This is not a hard requirement given the size of the credentials and the memory costs. Even in environments with high mobility requirements, caching the credentials of all the users in all the proxies or using caching algorithms are reasonable options. Finally, the concepts behind Proxychain can also be used in other domains with similar topology requirements. For example, remote authentication services such as RADIUS or DIAMETER, or authentication in IP Multimedia Subsystem (IMS) deployments could benefit from the performance, scalability and security advantages offered by Proxychain.

6.2 Security and Threat Analysis

Proxychain not only offers the security advantages of hash chains protocols (i.e., protection against eavesdropping and replay attacks), but also solves some of the weaknesses associated with these protocols [17]. For example, Proxychain provides integrity protection of the challenge in the form of an HMAC. This feature protects against an attacker located between the proxy and the client trying to change the counter (i) in the challenge to a lower value ($i = 1$) to obtain the complete hash chain sequence (small n attack [12]). Also, Proxychain provides mutual authentication between the UA and the proxy through the use of the session key generation based on a shared secret. The mutual authentication provided by Proxychain is less expensive and easier to implement than the one provided by protocols such as TLS or IPsec. In addition, Proxychain does not require hash chain synchronization[9] as S/Key does. The reason is that the hash chains are generated based on a secret derived from users' passwords.

Proxychain's threat model assumes that the database has a high level of security (a central database model facilitates this assumption). Only trusted entities (i.e., proxies) are allowed to communicate with the database using a robust security protocol (e.g., TLS or IPsec). Therefore, threats against the database can be considered low risks. In contrast, the proxies and the network traffic between proxies and UAs have a higher risk of being targeted by both, active and passive attackers.

An active attacker could try to compromise a proxy and steal its cached credentials. However, *Proxychain credentials cannot be used to impersonate users* (another advantage of hash chains). Instead, stolen Proxychain credentials could only be used to impersonate the proxy to the users due to the session key included in the credentials. In this scenario, only mutual authentication will be affected, resulting in the same security level provided by Digest authentication or S/Key (no server authentication). Therefore, an attacker will still need considerable effort to impersonate users even if she manages to steal the credentials cached by the proxy. While not implemented in our testbed, Proxychain can also include a revocation mechanism where the database can invalidate the credentials cached in a proxy. This mechanism will be useful in situations where a user needs to change her password or when a proxy has been compromised.

In addition, an active attacker could try to resend a previously captured user's response to make unauthorized requests in behalf of the user (replay attack). Proxychain provides a stronger defense against this type of attacks than Digest authentication due to the one-time password property of hash chains. Even if the attacker manages to capture a valid user's response (e.g., a MITM attack where the attacker prevents the user's response to reach the proxy), she will not be able to use it arbitrarily. The reason is that a Proxychain response includes the origin and destination of the request which are verified by the proxy (the attacker cannot modify the response because its integrity is protected).

Passive attackers (e.g., eavesdroppers) can monitor and record the communication between the UAs and the proxies. Proxychain protocol provides no additional information that passive attackers could use to impersonate users. Dictionary attacks against the challenge and the response values are still possible, but they require more effort than in Digest authentication due to the additional hash operations used in Proxychain.

Finally, Proxychain makes SIP authentication cheap enough to authenticate more than one message per session. Authenticating more SIP messages per session provides protection against several known attacks that target current SIP deployments. From the security perspective, all the messages should be authenticated to avoid vulnerabilities. Proxychain represents a first step in this direction.

6.3 Availability

The availability of the database is critical in scenarios with a central authentication service. For example, if the database becomes unavailable, the proxies will be unable to authenticate UAs requests. As a result, no call sessions can be established until the database is back online. This risk can be mitigated through mechanisms such as high availability clusters or backup sites. However, these alternatives are typically expensive and complex to manage.

Proxychain offers a cheaper alternative for database outages. The idea is that the database can create a list of authentication credentials with long enough hash chains and no expiration time. These backup-credentials can be stored offline in each proxy location and be activated when the database is not available. Once each proxy loads the backup-credentials in memory, they will be able to

authenticate UA requests as long as the credentials are active (sequence counter > 0). A naïve approach would be to generate backup-credentials with uniformly long hash chains (i.e., length = 1,000) to reduce the risks of users finishing their credentials before the database is back online. However, this approach is inefficient because very long hash chains will cause unnecessary overheads in the database and the UAs and lower performance during their generation. A more efficient approach would be to estimate the necessary length of the hash chains based on the expected time that the database is going to be unavailable. For example, a provider needs to install new hardware, requiring the database to be offline. The provider can estimate how many authenticated requests occur in a period of six hours based on its call statistics. For example, the provider can determine the call rate of its most active users. Assuming that the most active users make 10 calls per hour during busy hours, backup-credentials with a hash chain length of at least 60 will be required (also assuming that only one request per call is authenticated). Using Table 1, we know that the time to compute one credential with hash chain length = 100 is approximately 335 μsecs. Therefore, if the provider has 5 million users, the database will require approximately 28 minutes of computation to generate backup-credentials that will be active during 6 hours. This simple calculation could be made more robust by identifying those users most likely to far exceed the uses of the temporary credentials (i.e., profiling via long-term logging) and selectively increase the length of their hash chains.

7 Related Work

Authentication is a required service in most SIP deployments. The VoIP standard (RFC 3261 [21]) recommends the use of robust security mechanisms such as TLS, IPsec and S/MIME to provide authentication and other security guarantees. However, these mechanisms are computationally expensive [5, 6, 15, 16] and complex to manage (i.e., client certificates are required). Digest authentication, also recommended by RFC 3261, is more efficient and simpler authentication mechanism with lower implementation requirements than the previous schemes. As a result, it is the preferred authentication mechanism for most SIP deployments. However, previous research shows that Digest authentication can still have a considerable impact on the performance of a SIP infrastructure [19, 22], specially when a remote authentication database is employed [8]. In addition, in scenarios where the remote database is shared by multiple proxies, the database could become a bottleneck due to the high load the it receives. In this case, the database could be susceptible to DoS attacks. For example, multiple malicious clients could generate enough load to saturate the database, as was demonstrated by Traynor et al. [24] in cellular networks.

The impact on performance caused by authentication is also one of the reasons why only a few messages are authenticated in a SIP call transaction. This fact and the lack of mutual authentication lead to several possible attacks against a SIP infrastructure [2, 29]. Moreover, Digest authentication is considered a weak authentication protocol by today's cryptographic standards given its lack of mutual authentication and susceptibility to a number of other attacks [3, 9, 26, 27]. Several alternative schemes have been proposed to overcome the weaknesses of Digest authentication. Most of these alternatives focus on providing stronger security guarantees [4, 25, 27, 28], while others also focus on better performance [7]. However, these alternatives do not provide an implementation and experimental performance analysis to determine their impact on the performance of a SIP server.

To avoid the saturation of the database and improve performance, we present a new authentication mechanism based on temporary authentication vectors stored in the proxies. A similar idea is used in the Authentication and Key Agreement (AKA) [1] security protocol for 3G cellular networks. However, instead of using multiple authentication vectors as AKA does, our scheme uses a modified hash chain construction [13] to provide mutual authentication. Hash chains have been used in security protocols in different domains where efficiency is critical such as sensor networks [14, 20] and RFID tags [23]. Our work is the first to take advantage of the security, performance and space efficient properties of hash chains to reduce the overhead of the authentication process in SIP.

8 Conclusions

VoIP has and will continue to change telephony. These systems not only drastically reduce the costs associated with building and providing such services, but also offer the potential for rich new sets of features. Unfortunately, the large-scale usage of VoIP also creates a number of new security concerns. In this paper, we develop Proxychain, a mechanism that provides strong authentication between VoIP providers and their customers. Unlike previously deployed mechanisms, Proxychain is highly scalable and offers throughput improvements of greater than an order of magnitude. This increased efficiency allows providers not only to support a much larger customer base on a relatively limited hardware footprint, but also increases the overall security of the network by allowing for multiple message types to be authenticated. In so doing, we have significantly increased the robustness of VoIP systems.

Acknowledgments

We wish to thank Kevin Butler and our shepherd Galen Hunt for their valuable comments. This work was supported in part by the US National Science Foundation (CNS-0916047). Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 3GPP. ETSI Technical Specification 133 102 v7.1.0 - Security Architecture, 2006.

[2] Abdelnur, H., Avanesov, T., Rusinowitch, M., and State, R. Abusing SIP Authentication. In *Proceedings of the International Conference on Information Assurance and Security* (2008).

[3] Black, J., Cochran, M., and Highland, T. A Study of the MD5 Attacks: Insights and Improvements. In *Fast Software Encryption* (2006).

[4] Cao, F., and Jennings, C. Providing Response Identity and Authentication in IP Telephony. In *Proceedings of the First International Conference on Availability, Reliability and Security (ARES)* (2006).

[5] Cha, E.-C., Choi, H.-K., and Cho, S.-J. Evaluation of Security Protocols for the Session Initiation Protocol. In *Proceedings of 16th International Conference on Computer Communications and Networks (ICCCN)* (2007).

[6] Coarfa, C., Druschel, P., and Wallach, D. S. Performance analysis of TLS Web servers. *ACM Transactions on Computer Systems 24*, 1 (2006), 39–69.

[7] Cui, T., Gao, Q., and He, B. A lightweight authentication scheme for Session Initiation Protocol. In *International Conference on Communications, Circuits and Systems* (2008).

[8] Dacosta, I., Balasubramaniyan, V., Ahamad, M., and Traynor, P. Improving Authentication Performance of Distributed SIP Proxies. In *Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)* (2009).

[9] El Sawda, S., and Urien, P. SIP Security Attacks and Solutions: A state-of-the-art review. In *2nd International Conference on Information and Communication Technologies* (2006).

[10] Fisher, K., and Gruber, R. PADS: Processing arbitrary data streams. In *Proceedings of Workshop on Management and Processing of Data Streams* (2003), DIMACS.

[11] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication. *IETF RFC Editor* (1999).

[12] Kaufman, C., Perlman, R., and Speciner, M. *Network Security: Private Communication in a Public World*, second ed. Prentice Hall, 2002.

[13] Lamport, L. Password authentication with insecure communication. *Communications of the ACM 24* (1981), 770–772.

[14] Liu, D., and Ning, P. Multilevel μTESLA: Broadcast authentication for distributed sensor networks. *ACM Trans. Embed. Comput. Syst. 3*, 4 (2004), 800–836.

[15] Meenakshi, S., and Raghavan, S. Impact of IPSec Overhead on Web Application Servers. In *International Conference on Advanced Computing and Communications(ADCOM)* (2006).

[16] Miltchev, S., Ioannidis, S., and Keromytis, A. D. A Study of the Relative Costs of Network Security Protocols. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference* (2002).

[17] Mitchell, C. J., and Chen, L. Comments on the S/KEY user authentication scheme. *SIGOPS Oper. Syst. Rev. 30* (1996), 12–16.

[18] Mukerjee, S. Subscriber management for next-generation networks. `http://www.xchangemag.com/webexclusives/62h2815505.html`, 2006.

[19] Nahum, E. M., Tracey, J., and Wright, C. P. Evaluating SIP server performance. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (2007).

[20] Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J. D. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the International Conference on Mobile Computing and Networks (MOBICOM)* (2001).

[21] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. RFC 3261: SIP: Session Initiation Protocol. *IETF RFC Editor* (2002).

[22] Salsano, S., Veltri, L., and Papalilo, D. SIP security issues: the SIP authentication procedure and its processing load. *IEEE Network 16* (2002), 38–44.

[23] Syamsuddin, I., Dillon, T., Chang, E., and Han, S. A Survey of RFID Authentication Protocols Based on Hash-Chain Method. In *ICCIT: Proceedings of the Third International Conference on Convergence and Hybrid Information Technology* (2008).

[24] Traynor, P., Lin, M., Ongtang, M., Rao, V., Jaeger, T., La Porta, T., and McDaniel, P. On cellular botnets: Measuring the impact of malicious devices on a cellular network core. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2009).

[25] Wang, F., and Zhang, Y. A new provably secure authentication and key agreement mechanism for SIP using certificateless public-key cryptography. *Computer Communications 31* (2008), 7.

[26] Wang, X., and Yu, H. How to Break MD5 and Other Hash Functions. In *Proceedings of EUROCRYPT* (2005).

[27] Yang, C.-C., Wang, R.-C., and Liu, W.-T. Secure authentication scheme for session initiation protocol. *Computers and Security 24* (2005), 381–386.

[28] Yoon, E.-J., and Yoo, K.-Y. A New Authentication Scheme for Session Initiation Protocol. *Complex, Intelligent and Software Intensive Systems, International Conference* (2009).

[29] Zhang, R., Wang, X., Yang, X., and Jiang, X. Billing attacks on SIP-based VoIP systems. In *Proceedings of the first USENIX workshop on Offensive Technologies* (2007).

Notes

[1] Note that calls are placed through "Super-Nodes" in Skype, but that users sign on through a server located at 80.160.91.11.

[2] `http://www.netlab.cc.gatech.edu/`

[3] `https://www.planet-lab.org/`

[4] `http://www.opensips.org/`

[5] `http://www.mysql.com/`

[6] `http://sipp.sourceforge.net/`

[7] Based on empirical evaluation, not shown for space reasons.

[8] R^2 is the correlation coefficient, which indicates goodness of fit, with 0 being no match and 1 being perfect.

[9] Setting a new hash chain once the current one expires, using a secure secondary channel

ZooKeeper: Wait-free coordination for Internet-scale systems

Patrick Hunt and Mahadev Konar
Yahoo! Grid
{phunt,mahadev}@yahoo-inc.com

Flavio P. Junqueira and Benjamin Reed
Yahoo! Research
{fpj,breed}@yahoo-inc.com

Abstract

In this paper, we describe ZooKeeper, a service for coordinating processes of distributed applications. Since ZooKeeper is part of critical infrastructure, ZooKeeper aims to provide a simple and high performance kernel for building more complex coordination primitives at the client. It incorporates elements from group messaging, shared registers, and distributed lock services in a replicated, centralized service. The interface exposed by ZooKeeper has the wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems to provide a simple, yet powerful coordination service.

The ZooKeeper interface enables a high-performance service implementation. In addition to the wait-free property, ZooKeeper provides a per client guarantee of FIFO execution of requests and linearizability for all requests that change the ZooKeeper state. These design decisions enable the implementation of a high performance processing pipeline with read requests being satisfied by local servers. We show for the target workloads, 2:1 to 100:1 read to write ratio, that ZooKeeper can handle tens to hundreds of thousands of transactions per second. This performance allows ZooKeeper to be used extensively by client applications.

1 Introduction

Large-scale distributed applications require different forms of coordination. Configuration is one of the most basic forms of coordination. In its simplest form, configuration is just a list of operational parameters for the system processes, whereas more sophisticated systems have dynamic configuration parameters. Group membership and leader election are also common in distributed systems: often processes need to know which other processes are alive and what those processes are in charge of. Locks constitute a powerful coordination primitive that implement mutually exclusive access to critical resources.

One approach to coordination is to develop services for each of the different coordination needs. For example, Amazon Simple Queue Service [3] focuses specifically on queuing. Other services have been developed specifically for leader election [25] and configuration [27]. Services that implement more powerful primitives can be used to implement less powerful ones. For example, Chubby [6] is a locking service with strong synchronization guarantees. Locks can then be used to implement leader election, group membership, etc.

When designing our coordination service, we moved away from implementing specific primitives on the server side, and instead we opted for exposing an API that enables application developers to implement their own primitives. Such a choice led to the implementation of a *coordination kernel* that enables new primitives without requiring changes to the service core. This approach enables multiple forms of coordination adapted to the requirements of applications, instead of constraining developers to a fixed set of primitives.

When designing the API of ZooKeeper, we moved away from blocking primitives, such as locks. Blocking primitives for a coordination service can cause, among other problems, slow or faulty clients to impact negatively the performance of faster clients. The implementation of the service itself becomes more complicated if processing requests depends on responses and failure detection of other clients. Our system, Zookeeper, hence implements an API that manipulates simple *wait-free* data objects organized hierarchically as in file systems. In fact, the ZooKeeper API resembles the one of any other file system, and looking at just the API signatures, ZooKeeper seems to be Chubby without the lock methods, open, and close. Implementing wait-free data objects, however, differentiates ZooKeeper significantly from systems based on blocking primitives such as locks.

Although the wait-free property is important for performance and fault tolerance, it is not sufficient for coordination. We have also to provide order guarantees for operations. In particular, we have found that guaranteeing both *FIFO client ordering* of all operations and *linearizable writes* enables an efficient implementation of the service and it is sufficient to implement coordination primitives of interest to our applications. In fact, we can implement consensus for any number of processes with our API, and according to the hierarchy of Herlihy, ZooKeeper implements a universal object [14].

The ZooKeeper service comprises an ensemble of servers that use replication to achieve high availability and performance. Its high performance enables applications comprising a large number of processes to use such a coordination kernel to manage all aspects of coordination. We were able to implement ZooKeeper using a simple pipelined architecture that allows us to have hundreds or thousands of requests outstanding while still achieving low latency. Such a pipeline naturally enables the execution of operations from a single client in FIFO order. Guaranteeing FIFO client order enables clients to submit operations asynchronously. With asynchronous operations, a client is able to have multiple outstanding operations at a time. This feature is desirable, for example, when a new client becomes a leader and it has to manipulate metadata and update it accordingly. Without the possibility of multiple outstanding operations, the time of initialization can be of the order of seconds instead of sub-second.

To guarantee that update operations satisfy linearizability, we implement a leader-based atomic broadcast protocol [23], called Zab [24]. A typical workload of a ZooKeeper application, however, is dominated by read operations and it becomes desirable to scale read throughput. In ZooKeeper, servers process read operations locally, and we do not use Zab to totally order them.

Caching data on the client side is an important technique to increase the performance of reads. For example, it is useful for a process to cache the identifier of the current leader instead of probing ZooKeeper every time it needs to know the leader. ZooKeeper uses a watch mechanism to enable clients to cache data without managing the client cache directly. With this mechanism, a client can watch for an update to a given data object, and receive a notification upon an update. Chubby manages the client cache directly. It blocks updates to invalidate the caches of all clients caching the data being changed. Under this design, if any of these clients is slow or faulty, the update is delayed. Chubby uses leases to prevent a faulty client from blocking the system indefinitely. Leases, however, only bound the impact of slow or faulty clients, whereas ZooKeeper watches avoid the problem altogether.

In this paper we discuss our design and implementation of ZooKeeper. With ZooKeeper, we are able to implement all coordination primitives that our applications require, even though only writes are linearizable. To validate our approach we show how we implement some coordination primitives with ZooKeeper.

To summarize, in this paper our main contributions are:

Coordination kernel: We propose a wait-free coordination service with relaxed consistency guarantees for use in distributed systems. In particular, we describe our design and implementation of a *coordination kernel*, which we have used in many critical applications to implement various coordination techniques.

Coordination recipes: We show how ZooKeeper can be used to build higher level coordination primitives, even blocking and strongly consistent primitives, that are often used in distributed applications.

Experience with Coordination: We share some of the ways that we use ZooKeeper and evaluate its performance.

2 The ZooKeeper service

Clients submit requests to ZooKeeper through a client API using a ZooKeeper client library. In addition to exposing the ZooKeeper service interface through the client API, the client library also manages the network connections between the client and ZooKeeper servers.

In this section, we first provide a high-level view of the ZooKeeper service. We then discuss the API that clients use to interact with ZooKeeper.

Terminology. In this paper, we use *client* to denote a user of the ZooKeeper service, *server* to denote a process providing the ZooKeeper service, and *znode* to denote an in-memory data node in the ZooKeeper data, which is organized in a hierarchical namespace referred to as the *data tree*. We also use the terms update and write to refer to any operation that modifies the state of the data tree. Clients establish a *session* when they connect to ZooKeeper and obtain a session handle through which they issue requests.

2.1 Service overview

ZooKeeper provides to its clients the abstraction of a set of data nodes (znodes), organized according to a hierarchical name space. The znodes in this hierarchy are data objects that clients manipulate through the ZooKeeper API. Hierarchical name spaces are commonly used in file systems. It is a desirable way of organizing data objects, since users are used to this abstraction and it enables better organization of application meta-data. To refer to a given znode, we use the standard UNIX notation for file system paths. For example, we use `/A/B/C` to denote the path to znode `C`, where `C` has `B` as its parent and `B` has `A` as its parent. All znodes store data, and all znodes, except for ephemeral znodes, can have children.

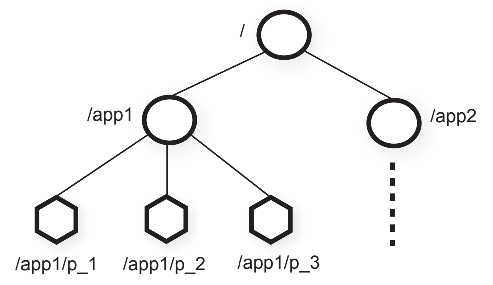


Figure 1: Illustration of ZooKeeper hierarchical name space.

There are two types of znodes that a client can create:

Regular: Clients manipulate regular znodes by creating and deleting them explicitly;

Ephemeral: Clients create such znodes, and they either delete them explicitly, or let the system remove them automatically when the session that creates them terminates (deliberately or due to a failure).

Additionally, when creating a new znode, a client can set a *sequential* flag. Nodes created with the sequential flag set have the value of a monotonically increasing counter appended to its name. If n is the new znode and p is the parent znode, then the sequence value of n is never smaller than the value in the name of any other sequential znode ever created under `p`.

ZooKeeper implements watches to allow clients to receive timely notifications of changes without requiring polling. When a client issues a read operation with a watch flag set, the operation completes as normal except that the server promises to notify the client when the information returned has changed. Watches are one-time triggers associated with a session; they are unregistered once triggered or the session closes. Watches indicate that a change has happened, but do not provide the change. For example, if a client issues a `getData(''/foo'', true)` before "/foo" is changed twice, the client will get one watch event telling the client that data for "/foo" has changed. Session events, such as connection loss events, are also sent to watch callbacks so that clients know that watch events may be delayed.

Data model. The data model of ZooKeeper is essentially a file system with a simplified API and only full data reads and writes, or a key/value table with hierarchical keys. The hierarchal namespace is useful for allocating subtrees for the namespace of different applications and for setting access rights to those subtrees. We also exploit the concept of directories on the client side to build higher level primitives as we will see in section 2.4.

Unlike files in file systems, znodes are not designed for general data storage. Instead, znodes map to abstractions of the client application, typically corresponding to meta-data used for coordination purposes. To illustrate, in Figure 1 we have two subtrees, one for Application 1 (`/app1`) and another for Application 2 (`/app2`). The subtree for Application 1 implements a simple group membership protocol: each client process p_i creates a znode `p_i` under `/app1`, which persists as long as the process is running.

Although znodes have not been designed for general data storage, ZooKeeper does allow clients to store some information that can be used for meta-data or configuration in a distributed computation. For example, in a leader-based application, it is useful for an application server that is just starting to learn which other server is currently the leader. To accomplish this goal, we can have the current leader write this information in a known location in the znode space. Znodes also have associated meta-data with time stamps and version counters, which allow clients to track changes to znodes and execute conditional updates based on the version of the znode.

Sessions. A client connects to ZooKeeper and initiates a session. Sessions have an associated timeout. ZooKeeper considers a client faulty if it does not receive anything from its session for more than that timeout. A session ends when clients explicitly close a session handle or ZooKeeper detects that a clients is faulty. Within a session, a client observes a succession of state changes that reflect the execution of its operations. Sessions enable a client to move transparently from one server to another within a ZooKeeper ensemble, and hence persist across ZooKeeper servers.

2.2 Client API

We present below a relevant subset of the ZooKeeper API, and discuss the semantics of each request.

`create(path, data, flags)`: Creates a znode with path name `path`, stores `data[]` in it, and returns the name of the new znode. `flags` enables a client to select the type of znode: regular, ephemeral, and set the sequential flag;

`delete(path, version)`: Deletes the znode `path` if that znode is at the expected version;

`exists(path, watch)`: Returns true if the znode with path name `path` exists, and returns false otherwise. The `watch` flag enables a client to set a watch on the znode;

`getData(path, watch)`: Returns the data and meta-data, such as version information, associated with the znode. The `watch` flag works in the same way as it does for `exists()`, except that ZooKeeper does not set the watch if the znode does not exist;

`setData(path, data, version)`: Writes `data[]` to znode `path` if the version number is the current version of the znode;

`getChildren(path, watch)`: Returns the set of names of the children of a znode;

`sync(path)`: Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to. The path is currently ignored.

All methods have both a synchronous and an asynchronous version available through the API. An application uses the synchronous API when it needs to execute a single ZooKeeper operation and it has no concurrent tasks to execute, so it makes the necessary ZooKeeper call and blocks. The asynchronous API, however, enables an application to have both multiple outstanding ZooKeeper operations and other tasks executed in parallel. The ZooKeeper client guarantees that the corresponding callbacks for each operation are invoked in order.

Note that ZooKeeper does not use handles to access znodes. Each request instead includes the full path of the znode being operated on. Not only does this choice simplifies the API (no `open()` or `close()` methods), but it also eliminates extra state that the server would need to maintain.

Each of the update methods take an expected version number, which enables the implementation of conditional updates. If the actual version number of the znode does not match the expected version number the update fails with an unexpected version error. If the version number is -1, it does not perform version checking.

2.3 ZooKeeper guarantees

ZooKeeper has two basic ordering guarantees:

Linearizable writes: all requests that update the state of ZooKeeper are serializable and respect precedence;

FIFO client order: all requests from a given client are executed in the order that they were sent by the client.

Note that our definition of linearizability is different from the one originally proposed by Herlihy [15], and we call it *A-linearizability* (asynchronous linearizability). In the original definition of linearizability by Herlihy, a client is only able to have one outstanding operation at a time (a client is one thread). In ours, we allow a client to have multiple outstanding operations, and consequently we can choose to guarantee no specific order for outstanding operations of the same client or to guarantee FIFO order. We choose the latter for our property. It is important to observe that all results that hold for linearizable objects also hold for A-linearizable objects because a system that satisfies A-linearizability also satisfies linearizability. Because only update requests are A-linearizable, ZooKeeper processes read requests locally at each replica. This allows the service to scale linearly as servers are added to the system.

To see how these two guarantees interact, consider the following scenario. A system comprising a number of processes elects a leader to command worker processes. When a new leader takes charge of the system, it must change a large number of configuration parameters and notify the other processes once it finishes. We then have two important requirements:

- As the new leader starts making changes, we do not want other processes to start using the configuration that is being changed;
- If the new leader dies before the configuration has been fully updated, we do not want the processes to use this partial configuration.

Observe that distributed locks, such as the locks provided by Chubby, would help with the first requirement but are insufficient for the second. With ZooKeeper, the new leader can designate a path as the *ready* znode; other processes will only use the configuration when that znode exists. The new leader makes the configuration change by deleting *ready*, updating the various configuration znodes, and creating *ready*. All of these changes can be pipelined and issued asynchronously to quickly update the configuration state. Although the latency of a change operation is of the order of 2 milliseconds, a new leader that must update 5000 different znodes will take 10 seconds if the requests are issued one after the other; by issuing the requests asynchronously the requests will take less than a second. Because of the ordering guarantees, if a process sees the *ready* znode, it must also see all the configuration changes made by the new leader. If the new leader dies before the *ready* znode is created, the other processes know that the configuration has not been finalized and do not use it.

The above scheme still has a problem: what happens if a process sees that *ready* exists before the new leader starts to make a change and then starts reading the configuration while the change is in progress. This problem is solved by the ordering guarantee for the notifications: if a client is watching for a change, the client will see the notification event before it sees the new state of the system after the change is made. Consequently, if the process that reads the *ready* znode requests to be notified of changes to that znode, it will see a notification informing the client of the change before it can read any of the new configuration.

Another problem can arise when clients have their own communication channels in addition to ZooKeeper. For example, consider two clients A and B that have a shared configuration in ZooKeeper and communicate through a shared communication channel. If A changes the shared configuration in ZooKeeper and tells B of the change through the shared communication channel, B would expect to see the change when it re-reads the configuration. If B's ZooKeeper replica is slightly behind A's, it may not see the new configuration. Using the above guarantees B can make sure that it sees the most up-to-date information by issuing a write before re-reading the configuration. To handle this scenario more efficiently ZooKeeper provides the `sync` request: when followed by a read, constitutes a *slow read*. `sync` causes a server to apply all pending write requests before processing the read without the overhead of a full write. This primitive is similar in idea to the `flush` primitive of ISIS [5].

ZooKeeper also has the following two liveness and durability guarantees: if a majority of ZooKeeper servers are active and communicating the service will be available; and if the ZooKeeper service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover.

2.4 Examples of primitives

In this section, we show how to use the ZooKeeper API to implement more powerful primitives. The ZooKeeper service knows nothing about these more powerful primitives since they are entirely implemented at the client using the ZooKeeper client API. Some common primitives such as group membership and configuration management are also wait-free. For others, such as rendezvous, clients need to wait for an event. Even though ZooKeeper is wait-free, we can implement efficient blocking primitives with ZooKeeper. ZooKeeper's ordering guarantees allow efficient reasoning about system state, and watches allow for efficient waiting.

Configuration Management ZooKeeper can be used to implement dynamic configuration in a distributed application. In its simplest form configuration is stored in a znode, z_c. Processes start up with the full pathname of z_c. Starting processes obtain their configuration by reading z_c with the watch flag set to true. If the configuration in z_c is ever updated, the processes are notified and read the new configuration, again setting the watch flag to true.

Note that in this scheme, as in most others that use watches, watches are used to make sure that a process has the most recent information. For example, if a process watching z_c is notified of a change to z_c and before it can issue a read for z_c there are three more changes to z_c, the process does not receive three more notification events. This does not affect the behavior of the process, since those three events would have simply notified the process of something it already knows: the information it has for z_c is stale.

Rendezvous Sometimes in distributed systems, it is not always clear a priori what the final system configuration will look like. For example, a client may want to start a master process and several worker processes, but the starting processes is done by a scheduler, so the client does not know ahead of time information such as addresses and ports that it can give the worker processes to connect to the master. We handle this scenario with ZooKeeper using a rendezvous znode, z_r, which is an node created by the client. The client passes the full pathname of z_r as a startup parameter of the master and worker processes. When the master starts it fills in z_r with information about addresses and ports it is using. When workers start, they read z_r with watch set to true. If z_r has not been filled in yet, the worker waits to be notified when z_r is updated. If z_r is an ephemeral node, master and worker processes can watch for z_r to be deleted and clean themselves up when the client ends.

Group Membership We take advantage of ephemeral nodes to implement group membership. Specifically, we use the fact that ephemeral nodes allow us to see the state of the session that created the node. We start by designating a znode, z_g to represent the group. When a process member of the group starts, it creates an ephemeral child znode under z_g. If each process has a unique name or identifier, then that name is used as the name of the child znode; otherwise, the process creates the znode with the SEQUENTIAL flag to obtain a unique name assignment. Processes may put process information in the data of the child znode, addresses and ports used by the process, for example.

After the child znode is created under z_g the process starts normally. It does not need to do anything else. If the process fails or ends, the znode that represents it under z_g is automatically removed.

Processes can obtain group information by simply listing the children of z_g. If a process wants to monitor changes in group membership, the process can set the watch flag to true and refresh the group information (always setting the watch flag to true) when change notifications are received.

Simple Locks Although ZooKeeper is not a lock service, it can be used to implement locks. Applications using ZooKeeper usually use synchronization primitives tailored to their needs, such as those shown above. Here we show how to implement locks with ZooKeeper to show that it can implement a wide variety of general synchronization primitives.

The simplest lock implementation uses "lock files". The lock is represented by a znode. To acquire a lock, a client tries to create the designated znode with the EPHEMERAL flag. If the create succeeds, the client holds the lock. Otherwise, the client can read the znode with the watch flag set to be notified if the current leader dies. A client releases the lock when it dies or explicitly deletes the znode. Other clients that are waiting for a lock try again to acquire a lock once they observe the znode being deleted.

While this simple locking protocol works, it does have some problems. First, it suffers from the herd effect. If there are many clients waiting to acquire a lock, they will all vie for the lock when it is released even though only one client can acquire the lock. Second, it only implements exclusive locking. The following two primitives show how both of these problems can be overcome.

Simple Locks without Herd Effect We define a lock znode l to implement such locks. Intuitively we line up all the clients requesting the lock and each client obtains the lock in order of request arrival. Thus, clients wishing to obtain the lock do the following:

Lock
```
1 n = create(l + "/lock-", EPHEMERAL|SEQUENTIAL)
2 C = getChildren(l, false)
3 if n is lowest znode in C, exit
4 p = znode in C ordered just before n
5 if exists(p, true) wait for watch event
6 goto 2
```

Unlock
```
1 delete(n)
```

The use of the SEQUENTIAL flag in line 1 of `Lock` orders the client's attempt to acquire the lock with respect to all other attempts. If the client's znode has the lowest sequence number at line 3, the client holds the lock. Otherwise, the client waits for deletion of the znode that either has the lock or will receive the lock before this client's znode. By only watching the znode that precedes the client's znode, we avoid the herd effect by only waking up one process when a lock is released or a lock request is abandoned. Once the znode being watched by the client goes away, the client must check if it now holds the lock. (The previous lock request may have been abandoned and there is a znode with a lower sequence number still waiting for or holding the lock.)

Releasing a lock is as simple as deleting the znode n that represents the lock request. By using the EPHEMERAL flag on creation, processes that crash will automatically cleanup any lock requests or release any locks that they may have.

In summary, this locking scheme has the following advantages:

1. The removal of a znode only causes one client to wake up, since each znode is watched by exactly one other client, so we do not have the herd effect;
2. There is no polling or timeouts;
3. Because of the way we have implemented locking, we can see by browsing the ZooKeeper data the amount of lock contention, break locks, and debug locking problems.

Read/Write Locks To implement read/write locks we change the lock procedure slightly and have separate read lock and write lock procedures. The unlock procedure is the same as the global lock case.

Write Lock
```
1  n = create(l + "/write-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if n is lowest znode in C, exit
4  p = znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 2
```

Read Lock
```
1  n = create(l + "/read-", EPHEMERAL|SEQUENTIAL)
2  C = getChildren(l, false)
3  if no write znodes lower than n in C, exit
4  p = write znode in C ordered just before n
5  if exists(p, true) wait for event
6  goto 3
```

This lock procedure varies slightly from the previous locks. Write locks differ only in naming. Since read locks may be shared, lines 3 and 4 vary slightly because only earlier write lock znodes prevent the client from obtaining a read lock. It may appear that we have a "herd effect" when there are several clients waiting for a read lock and get notified when the "write-" znode with the lower sequence number is deleted; in fact, this is a desired behavior, all those read clients should be released since they may now have the lock.

Double Barrier Double barriers enable clients to synchronize the beginning and the end of a computation. When enough processes, defined by the barrier threshold, have joined the barrier, processes start their computation and leave the barrier once they have finished. We represent a barrier in ZooKeeper with a znode, referred to as b. Every process p registers with b – by creating a znode as a child of b – on entry, and unregisters – removes the child – when it is ready to leave. Processes can enter the barrier when the number of child znodes of b exceeds the barrier threshold. Processes can leave the barrier when all of the processes have removed their children. We use watches to efficiently wait for enter and exit conditions to be satisfied. To enter, processes watch for the existence of a `ready` child of b that will be created by the process that causes the number of children to exceed the barrier threshold. To leave, processes watch for a particular child to disappear and only check the exit condition once that znode has been removed.

3 ZooKeeper Applications

We now describe some applications that use ZooKeeper, and explain briefly how they use it. We show the primitives of each example in **bold**.

The Fetching Service Crawling is an important part of a search engine, and Yahoo! crawls billions of Web documents. The Fetching Service (FS) is part of the Yahoo! crawler and it is currently in production. Essentially, it has master processes that command page-fetching processes. The master provides the fetchers with configuration, and the fetchers write back informing of their status and health. The main advantages of using ZooKeeper for FS are recovering from failures of masters, guaranteeing availability despite failures, and decoupling the clients from the servers, allowing them to direct their request to healthy servers by just reading their status from ZooKeeper. Thus, FS uses ZooKeeper mainly to manage **configuration metadata**, although it also uses ZooKeeper to elect masters (**leader election**).

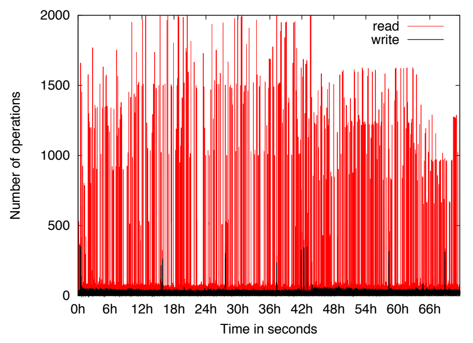


Figure 2: Workload for one ZK server with the Fetching Service. Each point represents a one-second sample.

Figure 2 shows the read and write traffic for a ZooKeeper server used by FS through a period of three days. To generate this graph, we count the number of operations for every second during the period, and each point corresponds to the number of operations in that second. We observe that the read traffic is much higher compared to the write traffic. During periods in which the rate is higher than $1,000$ operations per second, the read:write ratio varies between 10:1 and 100:1. The read operations in this workload are `getData()`, `getChildren()`, and `exists()`, in increasing order of prevalence.

Katta Katta [17] is a distributed indexer that uses ZooKeeper for coordination, and it is an example of a non-Yahoo! application. Katta divides the work of indexing using shards. A master server assigns shards to slaves and tracks progress. Slaves can fail, so the master must redistribute load as slaves come and go. The master can also fail, so other servers must be ready to take over in case of failure. Katta uses ZooKeeper to track the status of slave servers and the master (**group membership**), and to handle master failover (**leader election**). Katta also uses ZooKeeper to track and propagate the assignments of shards to slaves (**configuration management**).

Yahoo! Message Broker Yahoo! Message Broker (YMB) is a distributed publish-subscribe system. The system manages thousands of topics that clients can publish messages to and receive messages from. The topics are distributed among a set of servers to provide scalability. Each topic is replicated using a primary-backup scheme that ensures messages are replicated to two machines to ensure reliable message delivery. The servers that makeup YMB use a shared-nothing distributed architecture which makes coordination essential for correct operation. YMB uses ZooKeeper to manage the distribution of topics (**configuration metadata**), deal with failures of machines in the system (**failure detection** and **group membership**), and control system operation.

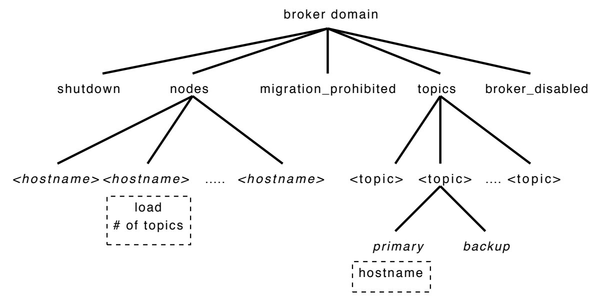


Figure 3: The layout of Yahoo! Message Broker (YMB) structures in ZooKeeper

Figure 3 shows part of the znode data layout for YMB. Each broker domain has a znode called `nodes` that has an ephemeral znode for each of the active servers that compose the YMB service. Each YMB server creates an ephemeral znode under `nodes` with load and status information providing both group membership and status information through ZooKeeper. Nodes such as `shutdown` and `migration_prohibited` are monitored by all of the servers that make up the service and allow centralized control of YMB. The `topics` directory has a child znode for each topic managed by YMB. These topic znodes have child znodes that indicate the primary and backup server for each topic along with the subscribers of that topic. The `primary` and `backup` server znodes not only allow servers to discover the servers in charge of a topic, but they also manage **leader election** and server crashes.

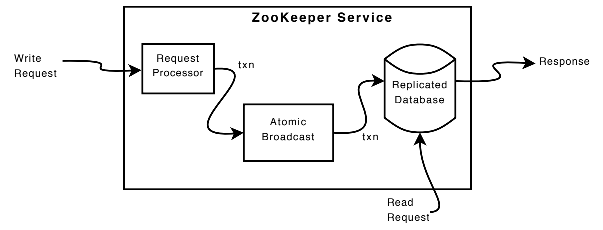


Figure 4: The components of the ZooKeeper service.

4 ZooKeeper Implementation

ZooKeeper provides high availability by replicating the ZooKeeper data on each server that composes the service. We assume that servers fail by crashing, and such faulty servers may later recover. Figure 4 shows the high-level components of the ZooKeeper service. Upon receiving a request, a server prepares it for execution (request processor). If such a request requires coordination among the servers (write requests), then they use an agreement protocol (an implementation of atomic broadcast), and finally servers commit changes to the ZooKeeper database fully replicated across all servers of the ensemble. In the case of read requests, a server simply reads the state of the local database and generates a response to the request.

The replicated database is an *in-memory* database containing the entire data tree. Each znode in the tree stores a maximum of 1MB of data by default, but this maximum value is a configuration parameter that can be changed in specific cases. For recoverability, we efficiently log updates to disk, and we force writes to be on the disk media before they are applied to the in-memory database. In fact, as Chubby [8], we keep a replay log (a write-ahead log, in our case) of committed operations and generate periodic snapshots of the in-memory database.

Every ZooKeeper server services clients. Clients connect to exactly one server to submit its requests. As we noted earlier, read requests are serviced from the local replica of each server database. Requests that change the state of the service, write requests, are processed by an agreement protocol.

As part of the agreement protocol write requests are forwarded to a single server, called the *leader*[1]. The rest of the ZooKeeper servers, called *followers*, receive message proposals consisting of state changes from the leader and agree upon state changes.

[1]Details of leaders and followers, as part of the agreement protocol, are out of the scope of this paper.

4.1 Request Processor

Since the messaging layer is atomic, we guarantee that the local replicas never diverge, although at any point in time some servers may have applied more transactions than others. Unlike the requests sent from clients, the transactions are *idempotent*. When the leader receives a write request, it calculates what the state of the system will be when the write is applied and transforms it into a transaction that captures this new state. The future state must be calculated because there may be outstanding transactions that have not yet been applied to the database. For example, if a client does a conditional `setData` and the version number in the request matches the future version number of the znode being updated, the service generates a `setDataTXN` that contains the new data, the new version number, and updated time stamps. If an error occurs, such as mismatched version numbers or the znode to be updated does not exist, an `errorTXN` is generated instead.

4.2 Atomic Broadcast

All requests that update ZooKeeper state are forwarded to the leader. The leader executes the request and broadcasts the change to the ZooKeeper state through Zab [24], an atomic broadcast protocol. The server that receives the client request responds to the client when it delivers the corresponding state change. Zab uses by default simple majority quorums to decide on a proposal, so Zab and thus ZooKeeper can only work if a majority of servers are correct (*i.e.*, with $2f + 1$ server we can tolerate f failures).

To achieve high throughput, ZooKeeper tries to keep the request processing pipeline full. It may have thousands of requests in different parts of the processing pipeline. Because state changes depend on the application of previous state changes, Zab provides stronger order guarantees than regular atomic broadcast. More specifically, Zab guarantees that changes broadcast by a leader are delivered in the order they were sent and all changes from previous leaders are delivered to an established leader before it broadcasts its own changes.

There are a few implementation details that simplify our implementation and give us excellent performance. We use TCP for our transport so message order is maintained by the network, which allows us to simplify our implementation. We use the leader chosen by Zab as the ZooKeeper leader, so that the same process that creates transactions also proposes them. We use the log to keep track of proposals as the write-ahead log for the in-

memory database, so that we do not have to write messages twice to disk.

During normal operation Zab does deliver all messages in order and exactly once, but since Zab does not persistently record the id of every message delivered, Zab may redeliver a message during recovery. Because we use idempotent transactions, multiple delivery is acceptable as long as they are delivered in order. In fact, ZooKeeper requires Zab to redeliver at least all messages that were delivered after the start of the last snapshot.

4.3 Replicated Database

Each replica has a copy in memory of the ZooKeeper state. When a ZooKeeper server recovers from a crash, it needs to recover this internal state. Replaying all delivered messages to recover state would take prohibitively long after running the server for a while, so ZooKeeper uses periodic snapshots and only requires redelivery of messages since the start of the snapshot. We call ZooKeeper snapshots *fuzzy snapshots* since we do not lock the ZooKeeper state to take the snapshot; instead, we do a depth first scan of the tree atomically reading each znode's data and meta-data and writing them to disk. Since the resulting fuzzy snapshot may have applied some subset of the state changes delivered during the generation of the snapshot, the result may not correspond to the state of ZooKeeper at any point in time. However, since state changes are idempotent, we can apply them twice as long as we apply the state changes in order.

For example, assume that in a ZooKeeper data tree two nodes `/foo` and `/goo` have values `f1` and `g1` respectively and both are at version 1 when the fuzzy snapshot begins, and the following stream of state changes arrive having the form ⟨`transactionType, path, value, new-version`⟩:

```
⟨SetDataTXN, /foo, f2, 2⟩
⟨SetDataTXN, /goo, g2, 2⟩
⟨SetDataTXN, /foo, f3, 3⟩
```

After processing these state changes, `/foo` and `/goo` have values `f3` and `g2` with versions 3 and 2 respectively. However, the fuzzy snapshot may have recorded that `/foo` and `/goo` have values `f3` and `g1` with versions 3 and 1 respectively, which was not a valid state of the ZooKeeper data tree. If the server crashes and recovers with this snapshot and Zab redelivers the state changes, the resulting state corresponds to the state of the service before the crash.

4.4 Client-Server Interactions

When a server processes a write request, it also sends out and clears notifications relative to any watch that corresponds to that update. Servers process writes in order and do not process other writes or reads concurrently. This ensures strict succession of notifications. Note that servers handle notifications locally. Only the server that a client is connected to tracks and triggers notifications for that client.

Read requests are handled locally at each server. Each read request is processed and tagged with a *zxid* that corresponds to the last transaction seen by the server. This *zxid* defines the partial order of the read requests with respect to the write requests. By processing reads locally, we obtain excellent read performance because it is just an in-memory operation on the local server, and there is no disk activity or agreement protocol to run. This design choice is key to achieving our goal of excellent performance with read-dominant workloads.

One drawback of using fast reads is not guaranteeing precedence order for read operations. That is, a read operation may return a stale value, even though a more recent update to the same znode has been committed. Not all of our applications require precedence order, but for applications that do require it, we have implemented `sync`. This primitive executes asynchronously and is ordered by the leader after all pending writes to its local replica. To guarantee that a given read operation returns the latest updated value, a client calls `sync` followed by the read operation. The FIFO order guarantee of client operations together with the global guarantee of `sync` enables the result of the read operation to reflect any changes that happened before the `sync` was issued. In our implementation, we do not need to atomically broadcast `sync` as we use a leader-based algorithm, and we simply place the `sync` operation at the end of the queue of requests between the leader and the server executing the call to `sync`. In order for this to work, the follower must be sure that the leader is still the leader. If there are pending transactions that commit, then the server does not suspect the leader. If the pending queue is empty, the leader needs to issue a null transaction to commit and orders the `sync` after that transaction. This has the nice property that when the leader is under load, no extra broadcast traffic is generated. In our implementation, timeouts are set such that leaders realize they are not leaders before followers abandon them, so we do not issue the null transaction.

ZooKeeper servers process requests from clients in FIFO order. Responses include the *zxid* that the response is relative to. Even heartbeat messages during intervals of no activity include the last *zxid* seen by the server that the client is connected to. If the client connects to a new server, that new server ensures that its view of the ZooKeeper data is at least as recent as the view of the client by checking the last *zxid* of the client against its last *zxid*. If the client has a more recent view than the server, the server does not reestablish the session with the client until the server has caught up. The client is guaranteed to be able to find another server that has a recent view of the system since the client only sees changes that have been replicated to a majority of the ZooKeeper servers. This behavior is important to guarantee durability.

To detect client session failures, ZooKeeper uses timeouts. The leader determines that there has been a failure if no other server receives anything from a client session within the session timeout. If the client sends requests frequently enough, then there is no need to send any other message. Otherwise, the client sends heartbeat messages during periods of low activity. If the client cannot communicate with a server to send a request or heartbeat, it connects to a different ZooKeeper server to re-establish its session. To prevent the session from timing out, the ZooKeeper client library sends a heartbeat after the session has been idle for $s/3$ ms and switch to a new server if it has not heard from a server for $2s/3$ ms, where s is the session timeout in milliseconds.

5 Evaluation

We performed all of our evaluation on a cluster of 50 servers. Each server has one Xeon dual-core 2.1GHz processor, 4GB of RAM, gigabit ethernet, and two SATA hard drives. We split the following discussion into two parts: throughput and latency of requests.

5.1 Throughput

To evaluate our system, we benchmark throughput when the system is saturated and the changes in throughput for various injected failures. We varied the number of servers that make up the ZooKeeper service, but always kept the number of clients the same. To simulate a large number of clients, we used 35 machines to simulate 250 simultaneous clients.

We have a Java implementation of the ZooKeeper server, and both Java and C clients[2]. For these experiments, we used the Java server configured to log to one dedicated disk and take snapshots on another. Our benchmark client uses the asynchronous Java client API, and each client has at least 100 requests outstanding. Each request consists of a read or write of 1K of data. We do not show benchmarks for other operations since the performance of all the operations that modify state are approximately the same, and the performance of non-state modifying operations, excluding `sync`, are approximately the same. (The performance of `sync` approximates that of a light-weight write, since the request must go to the leader, but does not get broadcast.) Clients send counts of the number of completed operations every $300ms$ and we sample every $6s$. To prevent memory overflows, servers throttle the number of concurrent requests in the system. ZooKeeper uses request throttling to keep servers from being overwhelmed. For these experiments, we configured the ZooKeeper servers to have a maximum of $2,000$ total requests in process.

[2]The implementation is publicly available at `http://hadoop.apache.org/zookeeper`.

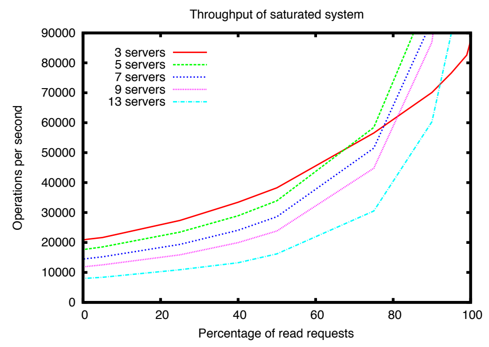


Figure 5: The throughput performance of a saturated system as the ratio of reads to writes vary.

Servers	100% Reads	0% Reads
13	460k	8k
9	296k	12k
7	257k	14k
5	165k	18k
3	87k	21k

Table 1: The throughput performance of the extremes of a saturated system.

In Figure 5, we show throughput as we vary the ratio of read to write requests, and each curve corresponds to a different number of servers providing the ZooKeeper service. Table 1 shows the numbers at the extremes of the read loads. Read throughput is higher than write throughput because reads do not use atomic broadcast. The graph also shows that the number of servers also has a negative impact on the performance of the broadcast protocol. From these graphs, we observe that the number of servers in the system does not only impact the number of failures that the service can handle, but also the workload the service can handle. Note that the curve for three servers crosses the others around 60%. This situation is not exclusive of the three-server configuration, and happens for all configurations due to the parallelism local reads enable. It is not observable for other configurations in the figure, however, because we have capped the maximum y-axis throughput for readability.

There are two reasons for write requests taking longer than read requests. First, write requests must go through atomic broadcast, which requires some extra processing and adds latency to requests. The other reason for longer processing of write requests is that servers must ensure that transactions are logged to non-volatile store before sending acknowledgments back to the leader. In principle, this requirement is excessive, but for our production systems we trade performance for reliability since ZooKeeper constitutes application ground truth. We use more servers to tolerate more faults. We increase write throughput by partitioning the ZooKeeper data into multiple ZooKeeper ensembles. This performance trade off between replication and partitioning has been previously observed by Gray *et al.* [12].

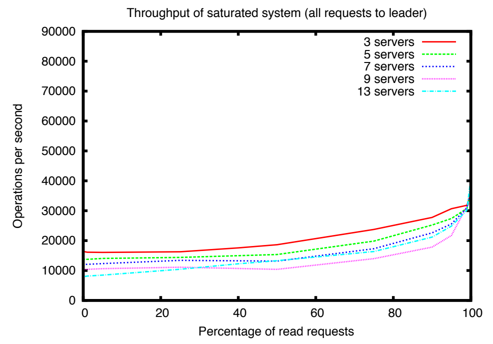


Figure 6: Throughput of a saturated system, varying the ratio of reads to writes when all clients connect to the leader.

ZooKeeper is able to achieve such high throughput by distributing load across the servers that makeup the service. We can distribute the load because of our relaxed consistency guarantees. Chubby clients instead direct all requests to the leader. Figure 6 shows what happens if we do not take advantage of this relaxation and forced the clients to only connect to the leader. As expected the throughput is much lower for read-dominant workloads, but even for write-dominant workloads the throughput is lower. The extra CPU and network load caused by servicing clients impacts the ability of the leader to coordinate the broadcast of the proposals, which in turn adversely impacts the overall write performance.

The atomic broadcast protocol does most of the work of the system and thus limits the performance of ZooKeeper more than any other component. Figure 7 shows the throughput of the atomic broadcast component. To benchmark its performance we simulate clients by generating the transactions directly at the leader, so there is no client connections or client requests and replies. At maximum throughput the atomic broadcast component becomes CPU bound. In theory the performance of Figure 7 would match the performance of ZooKeeper with 100% writes. However, the ZooKeeper client communication, ACL checks, and request to transaction conversions all require CPU. The contention for CPU lowers ZooKeeper throughput to substantially less than the atomic broadcast component in isolation. Because ZooKeeper is a critical production component, up to now our development focus for ZooKeeper has been correctness and robustness. There are plenty of opportunities for improving performance significantly by eliminating things like extra copies, multiple serializations of the same object, more efficient internal data structures, etc.

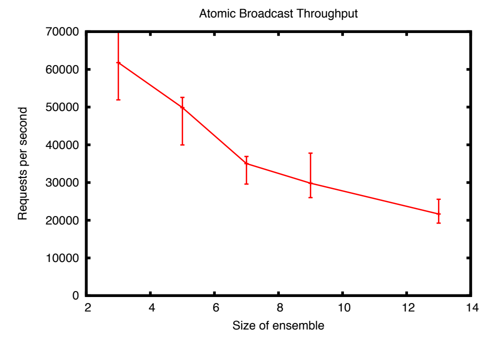


Figure 7: Average throughput of the atomic broadcast component in isolation. Error bars denote the minimum and maximum values.

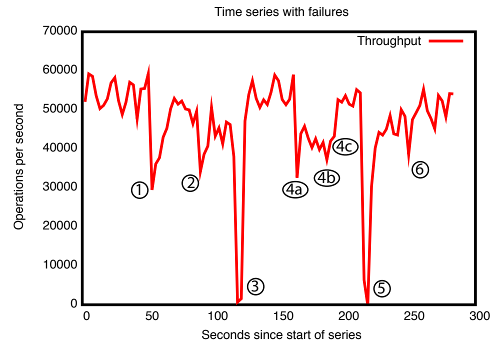


Figure 8: Throughput upon failures.

To show the behavior of the system over time as failures are injected we ran a ZooKeeper service made up of 5 machines. We ran the same saturation benchmark as before, but this time we kept the write percentage at a constant 30%, which is a conservative ratio of our expected workloads. Periodically we killed some of the server processes. Figure 8 shows the system throughput as it changes over time. The events marked in the figure are the following:

1. Failure and recovery of a follower;
2. Failure and recovery of a different follower;
3. Failure of the leader;
4. Failure of two followers (a, b) in the first two marks, and recovery at the third mark (c);
5. Failure of the leader.

6. Recovery of the leader.

There are a few important observations from this graph. First, if followers fail and recover quickly, then ZooKeeper is able to sustain a high throughput despite the failure. The failure of a single follower does not prevent servers from forming a quorum, and only reduces throughput roughly by the share of read requests that the server was processing before failing. Second, our leader election algorithm is able to recover fast enough to prevent throughput from dropping substantially. In our observations, ZooKeeper takes less than $200ms$ to elect a new leader. Thus, although servers stop serving requests for a fraction of second, we do not observe a throughput of zero due to our sampling period, which is on the order of seconds. Third, even if followers take more time to recover, ZooKeeper is able to raise throughput again once they start processing requests. One reason that we do not recover to the full throughput level after events 1, 2, and 4 is that the clients only switch followers when their connection to the follower is broken. Thus, after event 4 the clients do not redistribute themselves until the leader fails at events 3 and 5. In practice such imbalances work themselves out over time as clients come and go.

5.2 Latency of requests

To assess the latency of requests, we created a benchmark modeled after the Chubby benchmark [6]. We create a worker process that simply sends a create, waits for it to finish, sends an asynchronous delete of the new node, and then starts the next create. We vary the number of workers accordingly, and for each run, we have each worker create 50,000 nodes. We calculate the throughput by dividing the number of create requests completed by the total time it took for all the workers to complete.

	Number of servers			
Workers	3	5	7	9
1	776	748	758	711
10	2074	1832	1572	1540
20	2740	2336	1934	1890

Table 2: Create requests processed per second.

Table 2 show the results of our benchmark. The create requests include 1K of data, rather than 5 bytes in the Chubby benchmark, to better coincide with our expected use. Even with these larger requests, the throughput of ZooKeeper is more than 3 times higher than the published throughput of Chubby. The throughput of the single ZooKeeper worker benchmark indicates that the average request latency is 1.2ms for three servers and 1.4ms for 9 servers.

	# of clients		
# of barriers	**50**	**100**	**200**
200	9.4	19.8	41.0
400	16.4	34.1	62.0
800	28.9	55.9	112.1
1600	54.0	102.7	234.4

Table 3: Barrier experiment with time in seconds. Each point is the average of the time for each client to finish over five runs.

5.3 Performance of barriers

In this experiment, we execute a number of barriers sequentially to assess the performance of primitives implemented with ZooKeeper. For a given number of barriers b, each client first enters all b barriers, and then it leaves all b barriers in succession. As we use the double-barrier algorithm of Section 2.4, a client first waits for all other clients to execute the `enter()` procedure before moving to next call (similarly for `leave()`).

We report the results of our experiments in Table 3. In this experiment, we have 50, 100, and 200 clients entering a number b of barriers in succession, $b \in \{200, 400, 800, 1600\}$. Although an application can have thousands of ZooKeeper clients, quite often a much smaller subset participates in each coordination operation as clients are often grouped according to the specifics of the application.

Two interesting observations from this experiment are that the time to process all barriers increase roughly linearly with the number of barriers, showing that concurrent access to the same part of the data tree did not produce any unexpected delay, and that latency increases proportionally to the number of clients. This is a consequence of not saturating the ZooKeeper service. In fact, we observe that even with clients proceeding in lock-step, the throughput of barrier operations (enter and leave) is between 1,950 and 3,100 operations per second in all cases. In ZooKeeper operations, this corresponds to throughput values between 10,700 and 17,000 operations per second. As in our implementation we have a ratio of reads to writes of 4:1 (80% of read operations), the throughput our benchmark code uses is much lower compared to the raw throughput ZooKeeper can achieve (over 40,000 according to Figure 5). This is due to clients waiting on other clients.

6 Related work

ZooKeeper has the goal of providing a service that mitigates the problem of coordinating processes in distributed applications. To achieve this goal, its design uses ideas from previous coordination services, fault tolerant systems, distributed algorithms, and file systems.

We are not the first to propose a system for the coordination of distributed applications. Some early systems propose a distributed lock service for transactional applications [13], and for sharing information in clusters of computers [19]. More recently, Chubby proposes a system to manage advisory locks for distributed applications [6]. Chubby shares several of the goals of ZooKeeper. It also has a file-system-like interface, and it uses an agreement protocol to guarantee the consistency of the replicas. However, ZooKeeper is not a lock service. It can be used by clients to implement locks, but there are no lock operations in its API. Unlike Chubby, ZooKeeper allows clients to connect to any ZooKeeper server, not just the leader. ZooKeeper clients can use their local replicas to serve data and manage watches since its consistency model is much more relaxed than Chubby. This enables ZooKeeper to provide higher performance than Chubby, allowing applications to make more extensive use of ZooKeeper.

There have been fault-tolerant systems proposed in the literature with the goal of mitigating the problem of building fault-tolerant distributed applications. One early system is ISIS [5]. The ISIS system transforms abstract type specifications into fault-tolerant distributed objects, thus making fault-tolerance mechanisms transparent to users. Horus [30] and Ensemble [31] are systems that evolved from ISIS. ZooKeeper embraces the notion of virtual synchrony of ISIS. Finally, Totem guarantees total order of message delivery in an architecture that exploits hardware broadcasts of local area networks [22]. ZooKeeper works with a wide variety of network topologies which motivated us to rely on TCP connections between server processes and not assume any special topology or hardware features. We also do not expose any of the ensemble communication used internally in ZooKeeper.

One important technique for building fault-tolerant services is state-machine replication [26], and Paxos [20] is an algorithm that enables efficient implementations of replicated state-machines for asynchronous systems. We use an algorithm that shares some of the characteristics of Paxos, but that combines transaction logging needed for consensus with write-ahead logging needed for data tree recovery to enable an efficient implementation. There have been proposals of protocols for practical implementations of Byzantine-tolerant replicated state-machines [7, 10, 18, 1, 28]. ZooKeeper does not assume that servers can be Byzantine, but we do employ mechanisms such as checksums and sanity checks to catch non-malicious Byzantine faults. Clement *et al.* discuss an approach to make ZooKeeper fully Byzantine fault-tolerant without modifying the current server code base [9]. To date, we have not observed faults in production that would have been prevented using a fully Byzantine fault-tolerant protocol. [29].

Boxwood [21] is a system that uses distributed lock servers. Boxwood provides higher-level abstractions to applications, and it relies upon a distributed lock service based on Paxos. Like Boxwood, ZooKeeper is a component used to build distributed systems. ZooKeeper, however, has high-performance requirements and is used more extensively in client applications. ZooKeeper exposes lower-level primitives that applications use to implement higher-level primitives.

ZooKeeper resembles a small file system, but it only provides a small subset of the file system operations and adds functionality not present in most file systems such as ordering guarantees and conditional writes. ZooKeeper watches, however, are similar in spirit to the cache callbacks of AFS [16].

Sinfonia [2] introduces *mini-transactions*, a new paradigm for building scalable distributed systems. Sinfonia has been designed to store application data, whereas ZooKeeper stores application metadata. ZooKeeper keeps its state fully replicated and in memory for high performance and consistent latency. Our use of file system like operations and ordering enables functionality similar to mini-transactions. The znode is a convenient abstraction upon which we add watches, a functionality missing in Sinfonia. Dynamo [11] allows clients to get and put relatively small (less than 1M) amounts of data in a distributed key-value store. Unlike ZooKeeper, the key space in Dynamo is not hierarchal. Dynamo also does not provide strong durability and consistency guarantees for writes, but instead resolves conflicts on reads.

DepSpace [4] uses a tuple space to provide a Byzantine fault-tolerant service. Like ZooKeeper DepSpace uses a simple server interface to implement strong synchronization primitives at the client. While DepSpace's performance is much lower than ZooKeeper, it provides stronger fault tolerance and confidentiality guarantees.

7 Conclusions

ZooKeeper takes a wait-free approach to the problem of coordinating processes in distributed systems, by exposing wait-free objects to clients. We have found ZooKeeper to be useful for several applications inside and outside Yahoo!. ZooKeeper achieves throughput values of hundreds of thousands of operations per second for read-dominant workloads by using fast reads with watches, both of which served by local replicas. Although our consistency guarantees for reads and watches appear to be weak, we have shown with our use cases that this combination allows us to implement efficient and sophisticated coordination protocols at the client even though reads are not precedence-ordered and the implementation of data objects is wait-free. The wait-free property has proved to be essential for high performance.

Although we have described only a few applications, there are many others using ZooKeeper. We believe such a success is due to its simple interface and the powerful abstractions that one can implement through this interface. Further, because of the high-throughput of ZooKeeper, applications can make extensive use of it, not only course-grained locking.

Acknowledgements

We would like to thank Andrew Kornev and Runping Qi for their contributions to ZooKeeper; Zeke Huang and Mark Marchukov for valuable feedback; Brian Cooper and Laurence Ramontianu for their early contributions to ZooKeeper; Brian Bershad and Geoff Voelker made important comments on the presentation.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 59–74, New York, NY, USA, 2005. ACM.

[2] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, 2007.

[3] Amazon. Amazon simple queue service. `http://aws.amazon.com/sqs/`, 2008.

[4] A. N. Bessani, E. P. Alchieri, M. Correia, and J. da Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Systems Conference - EuroSys 2008*, Apr. 2008.

[5] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *SOSP '85: Proceedings of the 10th ACM symposium on Operating systems principles*, New York, USA, 1985. ACM Press.

[6] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[8] T. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th annual ACM symposium on Principles of distributed computing (PODC)*, Aug. 2007.

[9] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *Proceedings of the 22 nd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2009.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *SOSP '07: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007.

[11] 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: Proceedings of the 21st ACM symposium on Operating systems principles*, New York, NY, USA, 2007. ACM Press.

[12] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of SIGMOD '96*, pages 173–182, New York, NY, USA, 1996. ACM.

[13] A. Hastings. Distributed lock management in a transaction processing environment. In *Proceedings of IEEE 9th Symposium on Reliable Distributed Systems*, Oct. 1990.

[14] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1), 1991.

[15] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), July 1990.

[16] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1), 1988.

[17] Katta. Katta - distribute lucene indexes in a grid. `http://katta.wiki.sourceforge.net/`, 2008.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[19] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxclusters (extended abstract): a closely-coupled distributed system. *SIGOPS Oper. Syst. Rev.*, 19(5), 1985.

[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[21] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th ACM/USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004.

[22] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault. The totem system. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, June 1995.

[23] S. Mullender, editor. *Distributed Systems, 2nd edition*. ACM Press, New York, NY, USA, 1993.

[24] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, New York, NY, USA, 2008. ACM.

[25] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN*, 2008.

[26] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), 1990.

[27] A. Sherman, P. A. Lisiecki, A. Berkheimer, and J. Wein. ACMS: The Akamai configuration management system. In *NSDI*, 2005.

[28] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[29] Y. J. Song, F. Junqueira, and B. Reed. BFT for the skeptics. `http://www.net.t-labs.tu-berlin.de/~petr/BFTW3/abstracts/talk-abstract.pdf`.

[30] R. van Renesse and K. Birman. Horus, a flexible group communication systems. *Communications of the ACM*, 39(16), Apr. 1996.

[31] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Software - Practice and Experience*, 28(5), July 1998.

Testing Closed-Source Binary Device Drivers with DDT

Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

DDT is a system for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, resource leaks, etc. One can metaphorically think of it as a pesticide against device driver bugs. DDT combines virtualization with a specialized form of symbolic execution to thoroughly exercise tested drivers; a set of modular dynamic checkers identify bug conditions and produce detailed, executable traces for every path that leads to a failure. These traces can be used to easily reproduce and understand the bugs, thus both proving their existence and helping debug them. We applied DDT to several closed-source Microsoft-certified Windows device drivers and discovered 14 serious new bugs. DDT is easy to use, as it requires no access to source code and no assistance from users. We therefore envision DDT being useful not only to developers and testers, but also to consumers who want to avoid running buggy drivers in their OS kernels.

1 Introduction

Device drivers are one of the least reliable parts of an OS kernel. Drivers and other extensions—which comprise, for instance, 70% of the Linux operating system—have a reported error rate that is 3-7 times higher than the rest of the kernel code [11], making them substantially more failure-prone. Not surprisingly, 85% of Windows crashes are caused by driver failures [27]. Moreover, some drivers are vulnerable to malformed input from untrusted user-space applications, allowing an attacker to execute arbitrary code with kernel privilege [5].

It is therefore ironic that most computer users place full trust in closed-source binary device drivers: they run drivers (software that is often outsourced by hardware vendors to offshore programmers) inside the kernel at the highest privilege levels, yet enjoy a false sense of safety by purchasing anti-virus software and personal firewalls. Device driver flaws are more dangerous than application vulnerabilities, because device drivers can subvert the entire system and, by having direct memory access, can be used to overwrite both kernel and application memory. Recently, a zero-day vulnerability within a driver shipped with all versions of Windows allowed non-privileged users to elevate their privileges to Local System, leading to complete system compromise [24].

Our goal is to empower users to thoroughly test drivers before installing and loading them. We wish that the Windows pop-up requesting confirmation to install an uncertified driver also offered a "Test Now" button. By clicking that button, the user would launch a thorough test of the driver's binary; this could run locally or be automatically shipped to a trusted Internet service to perform the testing on behalf of the user. Such functionality would benefit not only end users, but also the IT staff charged with managing corporate networks, desktops, and servers using proprietary device drivers.

Our work applies to all drivers, including those for which source code is not available, thus complementing the existing body of driver reliability techniques. There exist several tools and techniques that can be used to build more reliable drivers [14, 23, 1] or to protect the kernel from misbehaving drivers [30], but these are primarily aimed at developers who have the driver's source code. Therefore, these techniques cannot be used (or even adapted) for the use of consumers on closed-source binary drivers. Our goal is to fill this gap.

We believe that the availability of consumer-side testing of device drivers is essential. As of 2004, there were 800,000 different kinds of PnP devices at customer sites, with 1,500 devices being added every day [26]. There were 31,000 unique drivers, and 9 new drivers were released every day. Each driver had ~3.5 versions in the field, with 88 new driver versions being released every day [26]. Faced with an increasing diversity of drivers, consumers (end users and IT specialists alike) need a way to perform end-to-end testing just before installation.

This paper describes DDT, a device driver testing system aimed at addressing these needs. DDT uses selective symbolic execution to explore the device driver's execution paths, and checks whether these paths can cause undesired behavior, such as crashing the kernel or overflowing a buffer. For each suspected case of bad behavior, DDT produces a replayable trace of the execution that led to the bug, providing the consumer irrefutable evidence of the problem. The trace can be re-executed on its own, or inside a debugger.

DDT currently works for Windows device drivers. We applied it to six popular binary drivers, finding 14 bugs with relatively little effort. These include race conditions, memory bugs, use of unchecked parameters, and resource leaks, all leading to kernel crashes or hangs. Since DDT found bugs in drivers that have successfully passed Microsoft certification, we believe it could be used to improve the driver certification process.

Our work makes two contributions: (1) The first system that can thoroughly and automatically test closed-source binary drivers, without access to the corresponding hardware device; (2) The concept of fully symbolic hardware—including symbolic interrupts—and demonstration of its use for testing kernel-mode binaries.

The rest of the paper is structured as follows: §2 provides a high-level overview of DDT, §3 describes the design in detail, §4 presents our current DDT prototype for Windows drivers, §5 evaluates DDT on six closed-source device drivers, §6 discusses limitations of our current prototype, §7 reviews related work, and §8 concludes.

2 Overview

DDT takes as input a binary device driver and outputs a report of found bugs, along with execution traces for each bug. The input driver is loaded in its native, unmodified environment, which consists of the OS kernel and the rest of the software stack above it. DDT then exercises automatically the driver along as many code paths as possible, and checks for undesired properties. When an error or misbehavior is detected, DDT logs the details of the path exploration along with an executable trace. This can be used for debugging, or merely as evidence to prove the presence of the bug.

DDT has two main components: a set of pluggable bug checkers and a driver exerciser (Figure 1). The exerciser is in charge of steering the driver down various execution paths—just like a personal trainer, it forces the driver to exercise all the various ways in which it can run. The dynamic checkers watch the execution and flag undesired driver behaviors along the executed paths. When they notice a bug, they ask the exerciser to produce information on how to reach that same situation again.

DDT provides a default set of checkers, and this set can be extended with an arbitrary number of other checkers for both safety and liveness properties (see §3.1). Currently, DDT detects the following types of bugs: memory access errors, including buffer overflows; race conditions and deadlocks; incorrectly handled interrupts; accesses to pageable memory when page faults are not allowed; memory leaks and other resource leaks; mishandled I/O requests (e.g., setting various I/O completion flags incorrectly); any action leading to kernel panic; and incorrect uses of kernel APIs.

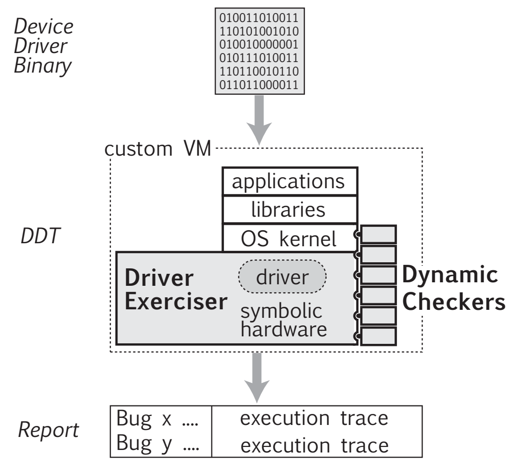


Figure 1: DDT's VM-based architecture.

These default checkers catch the majority of defects in the field. Ganapathi et al. found that the top driver problems causing crashes in Windows were 45% memory-related (e.g., bad pointers), 15% poorly handled exceptions, 13% infinite loops, and 3% unexpected traps [15]. A Microsoft report [26] found that, often, drivers crash the system due to not checking for error conditions following a call to the kernel. It is hypothesized that this is due to programmers copy-pasting code from the device driver development kit's succinct examples.

Black-box testing of closed-source binary device drivers is difficult and typically has low code coverage. This has two main reasons: First, it is hard to exercise the driver through the many layers of the software stack that lie between the driver's interface and the application interface. Second, closed-source programs are notoriously hard to test as a black box. The classic approach to testing such drivers is to try to produce inputs that exercise as many paths as possible and (perhaps) check for high-level properties (e.g., absence of kernel crashes) during those executions. Considering the wide range of possible inputs and system events that are hard to control (e.g., interrupts), this approach exercises relatively few paths, thus offering few opportunities to find bugs.

DDT uses selective symbolic execution [10] of the driver binary to automatically take the driver down as many paths as possible; the checkers verify desired properties along these paths. Symbolic execution [20, 6, 7] consists of providing a program with symbolic inputs (e.g., α or β) instead of concrete ones (e.g., 6 or "abc"), and letting these values propagate as the program executes, while tracking path constraints (e.g., $\beta = \alpha + 5$). When a symbolic value is used to decide the direction of a conditional branch, symbolic execution explores all feasible alternatives. On each branch, a suitable path constraint is added on the symbolic value to ensure its set of possible values satisfies the branch condition (e.g., $\beta < 0$ and $\beta >= 0$, respectively). Selective symbolic execution enables the symbolic execution of one piece of the software stack (the device driver, in our case) while the rest of the software runs concretely.

A key challenge is keeping the symbolic and the concrete portions of the execution synchronized. DDT supplies the driver with symbolic values on the calls from the kernel to the driver (§3.2) as well as on the returns from the hardware to the driver (§3.3), thus enabling an underlying symbolic execution engine to steer the driver on the various possible paths. When the driver returns values to a kernel-originated call, or when the driver calls into the kernel, parameters and driver are converted so that execution remains consistent, despite the alternation of symbolic and concrete execution.

DDT's *fully symbolic hardware* enables testing drivers even when the corresponding hardware device is not available. DDT never calls the actual hardware, but instead replaces all hardware reads with symbolic values, and discards all writes to hardware. Being able to test a driver without access to the hardware is useful, for example, for certification companies that cannot buy all the hardware variants for the drivers they test, or for consumers who would rather defer purchasing the device until they are convinced the driver is trustworthy.

Symbolic hardware also enables DDT to explore paths that are hard to test without simulators or specialized hardware. For example, many devices rely on interrupts to signal completion of operations to the device driver. DDT uses *symbolic interrupts* to inject such events at the various crucial points during the execution of the driver. Symbolic interrupts allow DDT to test different code interleavings and detect bugs like the race conditions described in §5.1.

DDT provides evidence of the bug and the means to debug it: a complete trace of the execution plus concrete inputs and system events that make the driver re-execute the buggy path in a regular, non-DDT environment.

3 Design

We now present DDT's design, starting with the types of bugs DDT looks for (§3.1), an overview of how drivers are exercised (§3.2), a description of fully symbolic hardware (§3.3), the use of annotations to extend DDT's capabilities (§3.4), and finally we show how generated traces are used to replay bugs and fix them (§3.5).

3.1 Detecting Undesired Behaviors

DDT uses two methods to detect failures along exercised paths: dynamic verification done by DDT's virtual machine (§3.1.1) and failure detection inside the guest OS (§3.1.2). VM-level checks are targeted at properties that require either instrumentation of driver code instructions or reasoning about multiple paths at a time. Guest OS-level checks leverage existing stress-testing and verification tools to catch bugs that require deeper knowledge of the kernel APIs. Most guest OS-level checks can be performed at the VM level as well, but it is often more convenient to write and deploy OS-level checkers.

3.1.1 Virtual Machine-Level Checks

Memory access verification in DDT is done at the VM level. On each memory access, DDT checks whether the driver has sufficient permissions to access that memory. For the purpose of access verification, DDT treats the following memory regions as accessible to drivers:

- Dynamically allocated memory and buffers;
- Buffers passed to the driver, such as network packets or strings from the Windows registry;
- Global kernel variables that are implicitly accessible to drivers;
- Current driver stack (accesses to memory locations below the stack pointer are prohibited, because these locations could be overwritten by an interrupt handler that saves context on the stack);
- Executable image area, i.e., loadable sections of the driver binary with corresponding permissions;
- Hardware-related memory areas (memory-mapped registers, DMA memory, or I/O ranges).

In order to track these memory regions, DDT hooks the kernel API functions and driver entry points. Every time the hooked functions are called, DDT analyzes their arguments to determine which memory was granted to (or revoked from) the driver. The required knowledge about specific kernel APIs can be provided through lightweight API annotations (see §3.4).

Beyond memory safety, DDT's simultaneous access to multiple execution paths (by virtue of employing symbolic execution) enables the implementation of bug detection techniques that reason about the code globally in terms of paths, such as infinite loop detection [34].

3.1.2 Guest Operating System-Level Checks

In addition to VM-level checkers, DDT can also reuse off-the-shelf runtime verification tools. These tools perform in-guest checking, oblivious to exactly how the driver is being driven along the observed execution paths. Since these tools are usually written by OS developers (e.g., for driver certification programs, like Microsoft's WHQL [25]), they can detect errors that require deep knowledge of the OS and its driver API.

When they find a bug, these dynamic tools typically crash the system to produce an error report containing a memory dump. DDT intercepts such premeditated crashes and reports the bug information to the user. DDT helps the runtime checkers find more bugs than they would do under normal concrete execution, because it symbolically execute the driver along many more paths.

DDT's modular architecture (Figure 1) allows reusing such tools without adaptation or porting. This means that driver developers' custom test suites can also be readily employed. Moreover, given DDT's design, such tools can be inserted at any level in the software stack, either in the form of device drivers or as software applications.

DDT can also automatically leverage kernel assertion checks, when they are present. For example, the checked build version of Windows contains many consistency checks—with DDT, these assertions get a better chance of being exercised along different paths.

3.2 Exercising the Driver: Kernel/Driver Interface

DDT implements selective symbolic execution [10], a technique for seamless transfer of system state between symbolic and concrete phases of execution. DDT obtaines similar properties to running the entire system symbolically, while in fact only running the driver symbolically. The transfer of state between phases is governed by a set of conversion hints (see §3.4). Using selective symbolic execution enables DDT to execute the driver within its actual environment, as opposed to requiring potentially incomplete models thereof [1, 6].

A typical driver is composed of several entry points. When the OS loads the driver, it calls its main entry point, similarly to a shell invoking the `main()` function of a program. This entry point registers with the kernel the driver's other entry points. For example, a typical driver would register `open`, `read`, `write`, and `close` entry points, which are then called by the OS when a user-mode application makes use of the driver.

When the kernel calls a driver's entry point, DDT transfers system state to a symbolic execution engine. It converts entry point arguments, and possibly other parts of concrete system state, to symbolic values, according to the annotations described in §3.4. For example, when the kernel calls the `SendPacket` function in a NIC driver, DDT makes the content of the network packet symbolic, to explore all the paths that depend on the packet's type.

When a driver calls a kernel function, DDT selects feasible values (at random) for its symbolic arguments. For example, if the driver calls the `AllocatePool` function with a symbolic length argument, DDT selects some concrete value *len* for the length that satisfies current constraints. However, this concretization subjects all subsequent paths to the constraint that length must equal *len*, and this may disable otherwise-feasible paths. Thus, DDT keeps track of all such concretization-related constraints and, if at some point in the future this constraint limits a choice of paths, DDT backtracks to the point of concretization, forks the entire machine state, and repeats the kernel call with different feasible concrete values, which could re-enable the presently unexplorable path.

To minimize overhead, DDT does concretization on-demand, i.e., delays it as long as possible by tracking symbolic values when executing in concrete mode and concretizing them only when they are actually read. This way, symbolic values that are not accessed by concretely running code are never concretized. In particular, all private driver state and buffers that are treated as opaque by the kernel end up being preserved in their symbolic form.

3.3 Exercising the Driver: Symbolic Hardware

DDT requires neither real hardware nor hardware models to test drivers—instead, DDT uses symbolic hardware. A symbolic device in DDT ignores all writes to its registers and produces symbolic values in response to reads. These symbolic values cause drivers to explore paths that depend on the device output.

Symbolic hardware produces symbolic interrupts, i.e., interrupts with a symbolic arrival time. Reasoning about interrupt arrival symbolically offers similar benefits to reasoning about program inputs symbolically: the majority of interrupt arrival times are equivalent to each other, so only one arrival time in each equivalence class need be produced. If a block of code does not read/write system state that is also read/written by the interrupt handler, then executing the interrupt handler at any point during the execution of that block has the same end result.

Currently, DDT implements a simplified model of symbolic interrupts. It symbolically delivers interrupts on each crossing of the kernel/driver boundary (i.e., before and after each kernel API call, and before and after each driver entry point execution). While not complete, we found that this strategy produces good results, because many important changes in driver state are related to crossing the kernel/driver interface.

Symbolic hardware with symbolic interrupts may force the driver on paths that are not possible in reality with correct hardware. For example, a symbolic interrupt may be issued after the driver instructed the device not to issue interrupts (e.g., by writing a control register). A correctly functioning device will therefore not deliver that interrupt. The natural solution would be to include the enabled/disabled interrupts status in the path constraints, and prevent interrupts from occurring when this is not possible. However, recent work [19] has shown that hardware often malfunctions, and that drivers must be sufficiently robust to handle such behavior anyway.

More generally, DDT's ability to test drivers against hardware failures is important, because chipsets often get

revised without the drivers being suitably updated. Consider a device that returns a value used by the driver as an array index. If the driver does not check the bounds (a common bug [19]) and a revised version of the chipset later returns a greater value, then the obsolete driver could experience an out-of-bounds error.

3.4 Enabling Rich Driver/Environment Interactions

Device drivers run at the bottom of the software stack, sandwiched between the kernel and hardware devices. The layers surrounding a driver are complex, and the different classes of device drivers use many different kernel subsystems. For instance, network, audio, and graphics drivers each use different kernel services and interfaces.

One may be tempted to run drivers in isolation for purposes of testing. Unfortunately, this requires an abstraction layer between the drivers and the rest of the stack, and building this layer is non-trivial. For example, testing a network driver would require the testbed to provide well-formed data structures when returning from a packet allocation function called by the driver.

DDT tests drivers by symbolically executing them in conjunction with the real kernel binary. By using the actual software stack (and thus the real kernel) instead of a simplified abstract model of it, DDT ensures that the device drivers get tested with the exact kernel behavior they would experience in reality. To this end, DDT needs to mediate the interactions with the layers around the driver in a way that keeps the symbolic execution of the driver consistent with the concrete execution of the kernel.

DDT performs various conversions between the symbolic and concrete domains. In its default mode, in which no annotations are used, DDT converts symbolic arguments passed to kernel functions into legal random concrete values and uses symbolic hardware, including symbolic interrupts. Driver entry point arguments are not touched. These conversions, however, can be fine-tuned by annotating API functions and driver entry points.

3.4.1 Extending DDT with Interface Annotations

DDT provides ways for developers to encode their knowledge of the driver/kernel API in annotations that improve DDT's achievable code coverage and bug finding abilities. Annotations allow DDT to detect not only low-level errors, but also logical bugs. Annotations are a one-time effort on the part of OS developers, testers, or a broader developer community.

The idea of encoding API usage rules in annotations is often used by model checking tools, with a recent notable example being SLAM [1]. However, DDT's annotations are lighter weight and substantially easier to write and keep up-to-date than the API models used by previous tools: preparing DDT annotations for the whole NDIS API, which consists of 277 exported functions, took about two weeks of on-and-off effort; preparing annotations for those 54 functions in the WDM API that were used by our sound drivers took one day.

DDT annotations are written in C and compiled to LLVM bitcode [22], which is then loaded by DDT at runtime and run in the context of QEMU-translated code, when necessary. The annotation code has direct access to, and control over, the guest system's state. Additionally, it can use a special API provided by DDT to create symbolic values and/or manipulate execution state.

The following annotation introduces positive integer symbolic values when the driver reads a configuration parameter from the Windows registry:

```
 1 void NdisReadConfiguration_return(CPU* cpu){
 2   if(*((PNDIS_STATUS) ARG(cpu, 0)) == 0
 3             && ARG(cpu, 4) == 1) {
 4     int symb = ddt_new_symb_int();
 5     if(symb >= 0)
 6       ((PNDIS_CONFIGURATION_PARAMETER)
 7         ARG(cpu, 1))->IntegerData = symb;
 8     else ddt_discard_state();
 9   }
10 }
```

This sample annotation function is invoked on the return path from `NdisReadConfiguration` (hence its name—line 1). It checks whether the call returned successfully (line 2) and whether the type of the value is integer (line 3). It then creates an unconstrained symbolic integer value using DDT's special API (line 4), after which it checks the value (line 5) and discards the path on which *symb* is not a positive integer (line 8).

DDT annotations fall into four categories:

Concrete-to-symbolic conversion hints apply to driver entry points' arguments and to return values from kernel functions called by the driver. They encode contracts about what constitute reasonable arguments or return values. For example, a memory allocation function can either return a valid pointer or a null pointer, so the annotation would instruct DDT to try both the originally returned concrete pointer, as well as the null-pointer alternative. The absence of this kind of conversion hints will cause DDT not to try all reasonable classes of values, which results solely in decreased coverage, i.e., false negatives.

Symbolic-to-concrete conversion hints specify the allowed set of values for arguments to kernel API functions called by drivers. They include various API usage rules that, if violated, may lead to crashes or data corruption. When a call to such an annotated function occurs, DDT verifies that all incorrect argument values are ruled out by the constraints on the current path; if not, it flags a potential bug. The absence of such annotations can lead DDT to concretize arguments into some values that are consistent with the path constraints (thus feasible in a real execution) but not uncover potential bugs (if the values happen to be OK according to the unspecified API usage rules). In other words, they can lead to false negatives, but not to false positives.

Resource allocation hints specify whether invoking an entry point or calling a kernel function grants or revokes the driver's access to any memory or other resources. This information is used to verify that the driver accesses only resources that the kernel explicitly allows it to access. It is also used to verify that all allocated resources are freed on exit paths. The absence of memory allocation hints can lead to false positives, but can be avoided, if necessary, by switching to a coarse-grained memory access verification scheme (as used, for instance, in Microsoft's Driver Verifier [25]).

Kernel crash handler hook: This annotation informs DDT of the address of the guest kernel's crash handler, as well as how to extract the crash information from memory. This annotation enables DDT to intercept all crashes when running the kernel concretely, such as the "blue screen of death" (BSOD) on Windows. This annotation is relied upon in our DDT prototype to cooperate with the Microsoft Driver Verifier's dynamic checkers.

3.4.2 Alternative Approaches

We have gone to great lengths to run the drivers in a real environment and avoid abstract modeling. Is it worth it?

One classic approach to ensuring device driver quality is stress-testing, which is how Microsoft certifies its third-party drivers [25]. However, this does not catch all bugs. As we shall see in the evaluation, even Microsoft-certified drivers shipped with Windows have bugs that cause the kernel to crash. However, powerful static analysis tools [1] can reason about corner-case conditions by abstracting the driver under test, without actually running it. Since static analysis does not run any code per se, it requires modeling the driver's environment.

We believe environment modeling generally does not scale, because kernels are large and evolve constantly. Modeling the kernel/driver API requires manual effort and is error prone. According to [1], developing around 60 API usage rules for testing Windows device drivers took more than three years. It also required many iterations of refinement based on false positives found during evaluation. In the end, the resulting models are only an approximation of the original kernel code, thus leading to both false negatives and, more importantly, false positives. A test tool that produces frequent false positives discourages developers from using it.

In contrast, we find DDT's annotations to be straightforward and easy to maintain. Moreover, if they are perceived by developers as too high of a burden, then DDT can be used in its default mode, without annotations.

Testing device drivers often requires access to either the physical device or a detailed model of it. For drivers that support several physical devices, testing must be repeated for each such device. In contrast, symbolic hardware enables not only testing drivers without a physical device, but also testing them against hardware bugs or corner cases that are hard to produce with a real device.

3.5 Verifying and Replaying Bugs

When DDT finishes testing a driver, it produces a detailed report containing all the bugs it found. This report consists of all faulty execution paths and contains enough information to accurately replay the execution, allowing the bug to be reproduced on the developer's or consumer's machine.

DDT's bug report is a collection of traces of the execution paths leading to the bugs. These traces contain the list of program counters of the executed instructions up to the bug occurrence, all memory accesses done by each instruction (address and value) and the type of the access (read or write). Traces contain information about creation and propagation of all symbolic values and constraints on branches taken. Each branch instruction has a flag indicating whether it forked execution or not, thus enabling DDT to subsequently reconstruct an execution tree of the explored paths; each node in the tree corresponds to a machine state. Finally, DDT associates with each failed path a set of concrete inputs and system events (e.g., interrupts) that take the driver along that path. The inputs are derived from the symbolic state by solving the corresponding path constraints [16, 7].

A DDT trace has enough information to replay the bug in the DDT VM. Each trace starts from an initial state (a "hibernated" snapshot of the system) and contains the exact sequence of instructions and memory accesses leading to the crash or hang. The traces are self-contained and directly executable. The size of these traces rarely exceeds 1 MB per bug, and usually they are much smaller. We believe DDT traces can easily be adapted to work with existing VM replay tools [13, 29, 21].

DDT also post-processes these traces off-line, to produce a palatable error report. DDT reconstructs the tree of execution paths and, for each leaf state that triggered a bug, it unwinds the execution path by traversing the execution tree to the root. Then it presents the corresponding execution path to the developer. When driver source code is available, DDT-produced execution paths can be automatically mapped to source code lines and variables, to help developers better visualize the buggy behavior.

For bugs leading to crashes, it is also possible to extract a Windows crash dump that can be analyzed with

WinDbg [25]—since each execution state maintained by DDT is a complete snapshot of the system, this includes the disk where the OS saved the crash dump. It is also worth noting that DDT execution traces can help debuggers go backwards through the buggy execution.

In theory, DDT traces could be directly executed outside the VM (e.g., in a debugger) using a natively executing OS, since the traces constitute slices through the driver code. The problem, though, is that the physical hardware would need to be coerced into providing the exact same sequence of interrupts as in the trace—perhaps this could be done with a PCI-based FPGA board that plays back a trace of interrupts. Another challenge is providing the same input and return values to kernel calls made by the driver—here DDT could leverage existing hooking techniques [4, 18] to intercept and modify these calls during replay. Finally, replaying on a real machine would involve triggering asynchronous events at points equivalent to those saved in the traces [33].

3.6 Analyzing Bugs

Execution traces produced by DDT can also help understand the cause of a bug. For example, if an assertion of a symbolic condition failed, execution traces can identify on what symbolic values the condition depended, when during the execution were they created, why they were created, and what concrete assignment of symbolic values would cause the assertion to fail. An assertion, bad pointer access, or a call that crashes the kernel might depend indirectly on symbolic values, due to control flow-based dependencies; most such cases are also identifiable in the execution traces.

Based on device specifications provided by hardware vendors, one can decide whether a bug can only occur when a device malfunctions. Say a DDT symbolic device returned a value that eventually led to a bug; if the set of possible concrete values implied by the constraints on that symbolic read does not intersect the set of possible values indicated by the specification, then one can safely conclude that the observed behavior would not have occurred unless the hardware malfunctioned.

One could write tools to automate the analysis and classification of bugs found by DDT, even though doing this manually is not hard. They could provide both user-readable messages, like "driver crashes in low-memory situations," and detailed technical information, like "`AllocateMemory` failed at location pc_1 caused a null pointer dereference at some other location pc_2.

4 DDT Implementation

We now describe our implementation of a DDT prototype for Windows device drivers (§4.1), which can be used by both developers and consumers to test binary drivers before installing them. We also show how to trick Windows into accepting DDT's symbolic hardware (§4.2) and how to identify and exercise the drivers' entry points (§4.3). Although Windows-specific, these techniques can be ported to other platforms as well.

4.1 DDT for Microsoft Windows

DDT uses a modified QEMU [2] machine emulator together with a modified version of the Klee symbolic execution engine [6]. DDT can run a complete, unmodified, binary software stack, comprising Windows, the drivers to be tested, and all associated applications.

4.1.1 Doing VM-Based Symbolic Execution

QEMU is an open-source machine emulator that supports many different processor architectures, like x86, SPARC, ARM, PowerPC, and MIPS. It emulates the CPU, memory, and devices using dynamic binary translation. QEMU's support of multiple architectures makes DDT available to more than just x86-based platforms.

DDT embeds an adapted version of Klee. To symbolically execute a program, one first compiles it to LLVM bitcode [22], which Klee can then interpret. Klee employs various constraint solving optimizations and coverage heuristics, which make it a good match for DDT.

To use Klee, we extended QEMU's back-end to generate LLVM bitcode. QEMU translates basic blocks from the guest CPU instruction set to a QEMU-specific intermediate representation—we translate from this intermediate representation to LLVM on the fly. The generated LLVM bitcode can be directly interpreted by Klee.

QEMU and Klee have different representations of program state, which have to be kept separate yet synchronized. In QEMU, the state is composed of the virtual CPU, VM physical memory, and various virtual devices. We encapsulate this data in Klee memory objects, and modified QEMU to use Klee's routines to manipulate the VM's physical memory. Thus, whenever the state of the CPU is changed (e.g., register written) or a device is accessed (e.g., interrupt controller registers are set), both QEMU and Klee see it, and Klee can perform symbolic execution in a consistent environment.

Symbolic execution generates path constraints that also have to be synchronized. Since QEMU and Klee keep a synchronized CPU, device, and memory state, any write to the state by one of them will be reflected in the path constraints kept by Klee. For example, when symbolically executing driver code accesses concrete kernel memory, it sees data consistent with its own execution so far. Conversely, when concrete code attempts to access a symbolic memory location, that location is automatically concretized, and a corresponding constraint is added to the current path. Data written by concrete code is seen as concrete by symbolically running driver code.

4.1.2 *Symbolic Execution of Driver Code*

QEMU runs in a loop, continuously fetching guest code blocks, translating them, and running them on the host CPU or in Klee. When a basic block is fetched, DDT checks whether the program counter is inside the driver of interest or not. If yes, QEMU generates a block of LLVM code (or fetches the code from a translation cache) and passes it to Klee; otherwise, it generates x86 machine code and sends it to the host processor.

DDT monitors kernel code execution and parses kernel data structures to detect driver load attempts. DDT catches the execution of the OS code responsible for invoking the load entry point of device drivers. For example, on Windows XP SP3, DDT monitors attempts to execute code at address `0x805A3990`, then parses the stack to fetch the device object. If the name of the driver corresponds to the one being monitored, DDT further parses the corresponding data structures to retrieve the code and data segment locations of the driver. Parsing the data structures is done transparently, by probing the virtual address space, without causing any side effects (e.g., no page faults are induced).

When the driver is executed with symbolic inputs, DDT forks execution paths as it encounters conditional branches. Forking consists primarily of making a copy of the contents of the CPU, the memory, and the devices, to make it possible to resume the execution from that state at a later time. In other words, each execution state consists conceptually of a complete system snapshot.

4.1.3 *Optimizing Symbolic Execution*

Since symbolic execution can produce large execution trees (exponential in the number of branches), DDT implements various optimizations to handle the large number of states generated by Klee. Moreover, each state is big, consisting of the entire physical memory and of the various devices (such as the contents of the virtual disk).

DDT uses chained copy-on-write: instead of copying the entire state upon an execution fork, DDT creates an empty memory object containing a pointer to the parent object. All subsequent writes place their values in the empty object, while reads that cannot be resolved locally (i.e., do not "hit" in the object) are forwarded up to the parent. Since quick forking can lead to deep state hierarchies, we cache each resolved read in the leaf state with a pointer to the target memory object, in order to avoid traversing long chains of pointers through parent objects.

4.1.4 *Symbolic Hardware*

For PCI devices, the OS allocates resources (memory, I/O regions, and interrupt line) for the device, as required the device descriptor, prior to loading the driver, and then writes the addresses of allocated resources to the device's registers. From that point, the device continuously monitors all memory accesses on the memory and I/O buses; when an address matches its allocated address range, the device handles the access. In QEMU, such accesses are handled by read/write functions specific to a each virtual device. For DDT symbolic devices, the write functions discard their arguments, and the read functions always returns an unconstrained symbolic value. When DDT decides to inject a symbolic interrupt, it calls the corresponding QEMU function to assert the right interrupt assigned to the symbolic device by the OS.

The execution of the driver also depends on certain parts of the device descriptor, not just on the device memory and I/O registers. For example, the descriptor may contain a hardware revision number that triggers slightly different behavior in the driver. Unfortunately, the device descriptor is parsed by the OS when selecting the driver and allocating device resources, so DDT cannot just make it symbolic. Instead, as the device drivers always accesses the descriptor through kernel API functions, we use annotations to insert appropriately constrained symbolic results when the driver reads the descriptor.

4.2 Fooling the OS into Accepting Symbolic Devices

Hardware buses like PCI and USB support Plug-and-Play, which is a set of mechanisms that modern operating systems use to detect insertion and removal of devices. The bus interface notifies the OS of such events. When the OS detects the presence of a new device, it loads the corresponding driver. The right driver is selected by reading the vendor and device ID of the inserted device. If the driver is for a PCI device, it will typically need to read the rest of the descriptor, i.e., the size of the register space and various I/O ranges.

DDT provides a PCI descriptor for a *fake device* to trick the OS into loading the driver to be tested. The fake device is an empty "shell" consisting of a descriptor containing the vendor and device IDs, as well as resource information. The fake device itself does not implement any logic other than producing symbolic values for read requests. Support for USB is similar: a USB descriptor pointing to a "shell" device is passed to the code implementing the bus, causing the target driver to be loaded.

Hardware descriptors are simple and can be readily obtained. If the actual hardware is available, the descriptors can be read directly from it. If the hardware is not present, it is possible to extract the information from pub-

Tested Driver	Size of Driver Binary File	Size of Driver Code Segment	Number of Functions in Driver	Number of Called Kernel Functions	Source Code Available ?
Intel Pro/1000	168 KB	120 KB	525	84	No
Intel Pro/100 (DDK)	70 KB	61 KB	116	67	Yes
Intel 82801AA AC97	39 KB	26 KB	132	32	No
Ensoniq AudioPCI	37 KB	23 KB	216	54	No
AMD PCNet	35 KB	28 KB	78	51	No
RTL8029	18 KB	14 KB	48	37	No

Table 1: Characteristics of Windows drivers used to evaluate DDT.

lic databases of hardware supported on Linux. If this information is not available, it can be extracted from the driver itself. For example, Windows drivers come with a `.inf` file specifying the vendor and device IDs of the supported devices. The device resources (e.g., memory or interrupt lines) are not directly available in the `.inf` files, but can be inferred after the driver is loaded, by watching for attempts to register the I/O space using OS APIs. We are working on a technique to automatically determine this information directly from the driver.

4.3 Exercising Driver Entry Points

DDT must detect that the OS has loaded a driver, determine the driver's entry points, coerce the OS into invoking them, and then symbolically execute them.

DDT automatically detects a driver's entry points by monitoring attempts of the driver to register such entry points with the kernel. Drivers usually export only one entry point, specified in the driver binary's file header. Upon invocation by the kernel, this routine fills data structures with entry point information and calls a registration function (e.g., `NdisMRegisterMiniport` for network drivers). In a similar way, DDT intercepts the registration of interrupt handlers.

DDT uses Microsoft's Device Path Exerciser as a concrete workload generator to invoke the entry points of the drivers to be tested. Device Path Exerciser is shipped with the Windows Driver Kit [25] and can be configured to invoke the entry points of a driver in various ways, testing both normal and error situations.

Each invoked entry point is symbolically executed by DDT. To accomplish this, DDT returns symbolic values on hardware register reads and, hooks various functions to inject symbolic data. Since execution can fork on branches within the driver, the execution can return to the OS through many different paths. To save memory and time, DDT terminates paths based on user-configurable criteria (e.g., if the entry point returns with a failure).

DDT attempts to maximize driver coverage using pluggable heuristics modules. The default heuristic attempts to maximize basic block coverage, similar to the one used in EXE [7]. It maintains a global counter for each basic block, indicating how many times the block was executed. The heuristic selects for the next execution step the basic block with the smallest value. This avoids states that are stuck, for instance, in polling loops (typical of device drivers). Depending on the driver, it is possible to choose different heuristics dynamically.

DDT tests for concurrency bugs by injecting symbolic interrupts before and after each kernel function called by the driver. It asserts the virtual interrupt line, causing QEMU to interrupt the execution of the current code and to invoke the OS's interrupt handler. The injection of symbolic interrupts is activated as soon as the target driver registers an interrupt handler for the device.

Drivers may legitimately access the kernel's data structures, and this must be taken into account by DDT, to avoid false reports of unauthorized memory accesses. First, drivers access global kernel variables, which must be explicitly imported by the driver; DDT scans the corresponding section of the loaded binary and grants the driver access to them. Second, private kernel data may be accessed via inlined functions (for example, NDIS drivers use macros that access kernel-defined private data fields in the `NDIS_PACKET` data structure). DDT provides annotations for identifying such data structures.

5 Evaluation

We applied DDT to six mature Microsoft-certified drivers—DDT found 14 serious bugs (§5.1). We also measured code coverage, and found that DDT achieves good coverage within minutes (§5.2). All reported measurements were done on an Intel 2 GHz Xeon CPU using 4 GB of RAM.

5.1 Effectiveness in Finding Bugs

We used DDT to test four network drivers and two sound card drivers, which use different Windows kernel APIs and are written in both C and C++ (Table 1). All drivers are reasonably sized, using tens of API functions; DDT scales well in this regard, mainly due to the fact that it needs no kernel API models. Most of these drivers have been tested by Microsoft as part of the WHQL certification process [25] and have been in use for many years.

DDT found bugs in all drivers we tested: memory leaks, memory corruptions, segmentation faults, and race

Tested Driver	Bug Type	Description
RTL8029	Resource leak	Driver does not always call `NdisCloseConfiguration` when initialization fails
RTL8029	Memory corruption	Driver does not check the range for `MaximumMulticastList` registry parameter
RTL8029	Race condition	Interrupt arriving before timer initialization leads to BSOD
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `QueryInformation`
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `SetInformation`
AMD PCNet	Resource leak	Driver does not free memory allocated with `NdisAllocateMemoryWithTag`
AMD PCNet	Resource leak	Driver does not free packets and buffers on failed initialization
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `ExAllocatePoolWithTag` returns NULL
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `PcNewInterruptSync` fails
Ensoniq AudioPCI	Race condition	Race condition in the initialization routine
Ensoniq AudioPCI	Race condition	Various race conditions with interrupts while playing audio
Intel Pro/1000	Memory leak	Memory leak on failed initialization
Intel Pro/100 (DDK)	Kernel crash	`KeReleaseSpinLock` called from DPC routine
Intel 82801AA AC97	Race condition	During playback, the interrupt handler can cause a BSOD

Table 2: Summary of previously unknown bugs discovered by DDT.

conditions. A summary of these findings is shown in Table 2, which shows *all* bug warnings issued by DDT, not just a subset. In particular, we encountered no false positives during testing.

The first two columns of the table are a direct output from DDT. Additionally, DDT produced execution traces that we manually analyzed (as per §3.6) in order to produce the last column of the table, explaining each bug. The analyses took a maximum of 20 minutes per bug. Testing each driver took a maximum of 4 hours, and this time includes adding missing API annotations and occasional debugging of the DDT prototype.

From among all bugs found by DDT, only one was related to improper hardware behavior: it was a subtle race condition in the RTL8029 driver, occurring right after the driver registered its interrupt handler, but before it initialized the timer routine and enabled interrupts on the device. If the interrupt fires at this point, the interrupt handler calls a kernel function to which it passes an uninitialized timer descriptor, causing a kernel crash. From the execution traces produced by DDT it was clear that the bug occurred in the driver interrupt handler routine after issuing a symbolic interrupt during driver initialization. We checked the address of the interrupt control register in the device documentation; since the execution traces contained no writes to that register, we concluded that the crash occurred before the driver enabled interrupts.

At the same time, if the device malfunctions and this bug manifests in the field, it is hard to imagine a way in which it could be fixed based on bug reports. It is hard to find this kind of bugs using classic stress-testing tools, even with malfunctioning hardware, because the interrupt might not be triggered by the hardware at exactly the right moment.

Another interesting bug involved memory corruption after parsing parameters (obtained from the registry) in the RTL8029 driver. The driver does not do any bounds checking when reading the *MaximumMulticastList* parameter during initialization. Later, the value of this parameter is used as an index into a fixed-size array. If the parameter has a large (or negative) value, memory corruption ensues and leads to a subsequent kernel panic. This explanation was easily obtained by looking at the execution traces: a faulty memory read was shown at an address equal to the sum of the base address returned by the memory allocator plus an unconstrained symbolic value injected when reading the registry.

An example of a common kind of bug is the incorrect handling of out-of-memory conditions during driver initialization. In the RTL8029, AMD PCNet, and Intel Pro/1000 drivers, such conditions lead to resource leaks: when memory allocation fails, the drivers do not release all the resources that were already allocated (heap memory, packet buffers, configuration handlers, etc.). In the Ensoniq AudioPCI driver, failed memory allocation leads to a segmentation fault, because the driver checks whether the memory allocation failed, but later uses the returned null pointer on an error handling path, despite the fact that the initial check failed.

An example of incorrectly used kernel API functions is a bug in the Intel Pro/100 driver. In its DPC (deferred procedure call) routine, the driver uses the `NdisReleaseSpinLock` function instead of `NdisDprReleaseSpinLock` (as it should for spinlocks acquired using `NdisDprAcquireSpinLock`). This is specifically prohibited by Microsoft documentation and in certain conditions can lead to setting the IRQ level to the wrong value, resulting in a kernel hang or panic.

We tried to find these bugs with the Microsoft Driver Verifier [25] running the driver concretely, but did not find any of them. Furthermore, since Driver Verifier crashes by default on the first bug found, looking for the next bug would typically require first fixing the found bug. In contrast, DDT finds multiple bugs in one run.

To assess the influence that annotations have on DDT's effectiveness, we re-tested these drivers with all

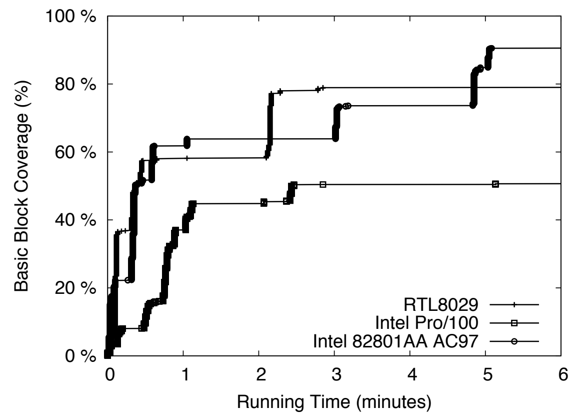


Figure 2: Relative coverage with time

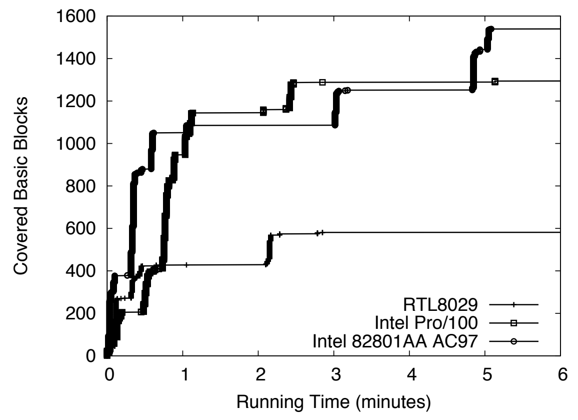


Figure 3: Absolute coverage with time

annotations turned off. We managed to reproduce all the race condition bugs, because their detection does not depend on the annotations. We also found the hardware-related bugs, cased by improper checks on hardware registers. However, removing the annotations resulted in decreased code coverage, so we did not find the memory leaks and the segmentation faults.

We initially wanted to compare DDT to the Microsoft SDV tool [1], a state-of-the-art static analysis tool for drivers. Since SDV requires source code, we used the Intel Pro/100 network card driver, whose source code appears in the Windows Drivers Development Kit. Unfortunately, we were not able to test this driver out-of-the-box using SDV, because the driver uses older versions of the NDIS API, that SDV cannot exercise. SDV also requires special entry point annotations in the source code, which were not present in the Intel Pro/100 driver. We resorted to comparing on the sample drivers shipped with SDV itself: SDV found the 8 sample bugs in 12 minutes, while DDT found all of them in 4 minutes.

We additionally injected several synthetic bugs in the sample driver (most of these hang the kernel): a deadlock, an out-of-order spinlock release, an extra release of a non-acquired spinlock, a "forgotten" unreleased spinlock, and a kernel call at the wrong IRQ level. SDV did not find the first 3 bugs, it found the last 2, and produced 1 false positive. DDT found all 5 bugs and no false positives in less than a third of the time that SDV ran.

We conclude that DDT can test drivers that existing tools cannot handle, and can find more subtle bugs in mature device drivers. In the next section, we evaluate the efficiency of DDT and assess its scalability.

5.2 Efficiency and Scalability

We evaluated DDT on drivers ranging in size from 18 KB to 168 KB. In Figure 2 we show how code coverage (as a fraction of total basic blocks) varied with time for a representative subset of the six drivers we tested. In Figure 3 we show absolute coverage in terms of number of basic blocks. We ran DDT until no more basic blocks were discovered for some amount of time. In all cases, a small number of minutes were sufficient to find the bugs we reported. For the network drivers, the workload consisted of sending one packet. For the audio drivers, we played a small sound file. DDT's symbolic execution explored paths starting from the exercised entry points. For more complex drivers, workload can be generated with the Device Path Exerciser (described in §4).

DDT has reasonable memory requirements. While testing the drivers in Table 1, DDT used at most 4 GB of memory, which is the current prototype's upper limit.

The coverage graphs show long flat periods of execution during which no new basic blocks are covered. These periods are delimited by the invocation of new entry points. The explanation is that the driver-loading phase triggers the execution of many new basic blocks, resulting in a first step. Then, more paths are exercised in it, without covering new blocks. Finally, the execution moves to another entry point, and so on. Eventually, no new entry points are exercised, and the curves flatten.

Overall, the results show that high coverage of binary drivers can be achieved automatically in just a few minutes. This suggests that DDT can be productively used even by end users on their home machines.

6 Discussion

Having seen that DDT is able to automatically find bugs in a reasonable amount of time, we now discuss some of DDT's limitations (§6.1) and the tradeoffs involved in testing binary drivers instead of their source code (§6.2).

6.1 Limitations

DDT subsumes several powerful driver testing tools, but still has limitations, which arise both from our design choices, as well as from technical limitations of the building blocks we use in the DDT prototype.

DDT uses symbolic execution, which is subject to the path explosion problem [3]. In the worst case, the number of states is exponential in the number of covered branches, and this can lead to high memory consumption and long running times for very large drivers. Moreover, solving path constraints at each branch is CPU-intensive. This limits DDT's ability to achieve good coverage for large drivers. We are exploring ways to mitigate this problem by running symbolic execution in parallel [12], and we are developing techniques for trimming the large space of paths to be explored [10]. Any improvements in the scalability of symbolic execution automatically improve DDT's coverage for very large drivers.

Like any bug finding tool, DDT might have false negatives. There are two causes for this: not checking for a specific kind of bug, or not covering a path leading to the bug. Since DDT can reuse any existing dynamic bug finding tool (by running it inside the virtual machine along all explored paths) and can be extended with other types of checkers, we expect that DDT can evolve over time into a tool that achieves superior test completeness.

Since DDT does not use real hardware and knows little about its expected behavior, DDT may find bugs that can only be triggered by a malfunctioning device. Even though it has been argued that such cases must be handled in high-reliability drivers [19], for some domains this may be too much overhead. In such cases, these false positives can be weeded out by looking at the execution traces, or by adding device-specific annotations.

Some driver entry points are triggered only when certain conditions hold deep within the execution tree. For example, the `TransferData` entry point in an NDIS driver is typically called when the driver receives a packet *and* provides some look-ahead data from it to the kernel *and* the kernel finds a driver that claims that packet. Since the packet contains purely symbolic data, and is concretized randomly when the kernel reads it, the likelihood of invoking the required handler is low. Annotating the function transmitting the look-ahead data to the kernel can solve this problem.

While testing drivers with DDT can be completely automated, our current DDT prototype requires some manual effort. A developer must provide DDT with PCI device information for the driver's device, install the driver inside a VM, and configure Microsoft Driver Verifier and a concrete workload generator. Once DDT runs, its output is a list of bugs and corresponding execution traces; the developer can optionally analyze the execution traces to find the cause of the encountered bugs. Even though this limits DDT's immediate usefulness to end users, DDT can be used today by hardware vendors to test drivers before releasing them, by OS vendors to certify drivers, and by system integrators to test final products before deployment.

DDT does not yet support USB, AGP, and PCI-express devices, partly due to the lack of such support in QEMU. This limitation prevents DDT from loading the drivers, but can be overcome by extending QEMU.

Finally, DDT currently has only a 32-bit implementation. This prevents DDT from using more than 4 GB of memory, thus limiting the number of explored paths. Although we implemented various optimizations, like swapping out unnecessary states to disk, memory is eventually exhausted. We ported Klee to 64-bit architectures and contributed it to the Klee mainline; we intend to port DDT as well.

6.2 Source-Level vs. Binary-Level Testing

DDT is a binary-level testing tool, and this has both benefits and drawbacks.

A binary tool can test the end result of a complex build tool chain. Device drivers are built with special compilers and linked to specific libraries. A miscompilation (e.g., a wrong field alignment in the data structures), or linking problems (e.g., a wrong library version), can be more easily detected by a binary testing tool.

Binary drivers, however, have meager typing information. The only types used by binaries are integers and pointers, and it may be possible to infer some types by looking at the API functions for which we have parameter information. Nevertheless, it is hard to find type-related bugs that do not result in abnormal operations. For example, a cast of a color value from the framebuffer to the wrong size could result in incorrect colors. Such bugs are more easily detected by source code analyzers.

7 Related Work

Two main approaches have been taken to improve the safety and reliability of device drivers. Offline approaches, like testing, concentrate on finding bugs before the driver is shipped. However, thoroughly testing a driver to the point of guaranteeing the absence of bugs is still economically infeasible, so bugs frequently make their way to the field. Online approaches aim to protect systems from bugs that are missed during the testing phase, but typically at the expense of runtime overhead and/or modifications to the OS kernel.

Testing device drivers can be done statically or dynamically. For example, SLAM [1] statically checks the source code of a device driver for correct Windows API usage. It uses a form of model checking combined with an abstract representation of the source code, suitable for the properties to be checked. However, it is subject to false positives and false negatives stemming from incomplete and/or imprecise API models.

Microsoft provides various tools for stress-testing device drivers running in their real environment. For example, Driver Verifier [25] provides deep testing of running device drivers, but it can miss rarely executed code paths. DDT combines the power of both static and dynamic tools: it runs drivers in a real environment, and combines its own checks with those of the Driver Verifier. Moreover, DDT employs fully symbolic hardware, leading to a more thorough exploration.

When testing is not enough, it is possible to continuously monitor the drivers at runtime and provide information on the cause of the crashes. For example, Nooks [31] combines in-kernel wrapping and hardware-enforced protection domains to trap common faults and recover from them. Nooks works with existing drivers, but requires source code and incurs runtime overhead.

SFI [32] and XFI [14] use faster software isolation techniques and provide fine grained isolation of the drivers to protect the kernel from references outside their logical protection domain. However, it can only protect against memory failures and incurs runtime overhead. XFI can work on binary drivers but still requires debugging information for the binaries in order to reliably disassemble them. SafeDrive [35] uses developer provided annotations to enforce type-safety constraints and system invariants for kernel-mode drivers written in C. Finally, BGI [8] provides byte-granularity memory protection to sandbox kernel extensions. BGI was also able to find driver bugs that manifest when running the drivers with BGI isolation. However BGI also requires access to the source code and incurs runtime overhead.

Minix [17] explicitly isolate drivers by running them in distinct address spaces; this approach is suitable for microkernels. Vino [28] introduces an alternative OS design, which combines software fault isolation with a lightweight transactional system to protect against large classes of problems in kernel extensions.

The idea of replacing reads from hardware with symbolic values has been mentioned before [3]. With DDT, we introduce the new concept of fully symbolic hardware, which can interact both with concretely running OSes and with symbolically running device drivers. Fully symbolic hardware can also issue symbolic interrupts, enabling the testing of various interleavings of device driver code and interrupt handlers.

Selective symbolic execution was first introduced in [10] and later reused in [9]. DDT shares common ideas with these, but is also distinguished by several aspects.

First, reverse engineering of a driver with RevNIC does not require execution to be sound. For example, RevNIC overwrites with unconstrained symbolic values the concrete parameters passed by the OS to the driver. In contrast, since DDT is a testing tool, it requires the execution to be sound to avoid false positives. This introduces additional requirements on injection of symbolic values and on concretization. For example, the concrete packet size must be replaced by a symbolic value constrained not to be greater than the original value, to avoid buffer overflows.

Second, DDT introduces the use of lightweight API annotations to describe the interface between a driver and a kernel. Annotations encode developers' knowledge about a specific kernel API, and help improve code coverage as well as detect more logic bugs. Such annotations were not present in RevNIC.

Third, DDT mixes in-VM instrumentation (bug checking) with instrumentation from outside the VM. DDT can reuse existing bug-finding tools that run in the guest OS, extending these tools with symbolic execution to work on multiple paths.

Finally, during symbolic execution, RevNIC only gathers executed LLVM code and traces of device accesses. In contrast, DDT analyzes the execution in order to track the origin of symbolic values and control flow dependencies through the path leading to a bug. DDT generates annotated execution traces and input values that help developers reproduce and understand the bugs.

8 Conclusion

We presented DDT, a tool for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, and resource leaks. We evaluated DDT on six mature Windows drivers and found 14 serious bugs that can cause a system to freeze or crash.

DDT combines virtualization with selective symbolic execution to thoroughly exercise tested drivers. A set of modular dynamic checkers identify bug conditions and produce detailed, executable traces for every path that leads to a failure. We showed how these traces can be used to provide evidence of the found bugs, as well as help understand and fix them.

DDT does not require access to source code and needs no assistance from users, thus making it widely applicable. We envision DDT being used by IT staff responsible for the reliability and security of desktops and servers, by OS vendors and system integrators, as well as by consumers who wish to avoid running buggy drivers in their operating system kernels.

Acknowledgments

We are indebted to our shepherd, Paul Barham, the anonymous USENIX reviewers, and our EPFL colleagues for their great help in improving our paper.

References

[1] 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/EuroSys European Conf. on Computer Systems*, 2006.

[2] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf.*, 2005.

[3] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[4] P. P. Bungale and C.-K. Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Intl. Conf. on Virtual Execution Environments*, 2007.

[5] S. Butt, V. Ganapathy, M. M. Swift, and C.-C. Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *Annual Computer Security Applications Conf.*, 2009.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[7] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communication Security*, 2006.

[8] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast byte-granularity software fault isolation. In *Symp. on Operating Systems Principles*, 2009.

[9] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems*, 2010.

[10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems*, 2009.

[11] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. 2001.

[12] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware*, 2009.

[13] G. W. Dunlap, D. Lucchetti, P. M. Chen, and M. Fetterman. Execution replay on multiprocessor virtual machines. In *Intl. Conf. on Virtual Execution Environments*, 2008.

[14] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[15] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. 2006.

[16] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Conf. on Programming Language Design and Implementation*, 2005.

[17] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Fault isolation for device drivers. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[18] G. Hoglund and J. Butler. *Rootkits: subverting the Windows Kernel*. Campus Press, 2006.

[19] A. Kadav, M. J. Renzelmann, and M. M. Swift. Tolerating hardware device failures in software. In *Symp. on Operating Systems Principles*, 2009.

[20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

[21] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf.*, 2005.

[22] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[23] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Symp. on Operating Systems Design and Implementation*, 2000.

[24] Microsoft security advisory #944653: Vulnerability in Macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[25] Microsoft. Windows logo program. http://www.microsoft.com/whdc.

[26] B. Murphy. Automating software failure reporting. *ACM Queue*, 2(8), 2004.

[27] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[28] M. Seltzer, Y. Endo, C. Small, and K. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symp. on Operating Systems Design and Implementation*, 1996.

[29] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conf.*, 2004.

[30] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4), 2006.

[31] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1), 2005.

[32] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Symp. on Operating Systems Principles*, 1993.

[33] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Conf. on Programming Language Design and Implementation*, 2008.

[34] J. Zhang. A path-based approach to the detection of infinite looping. In *Asia-Pacific Conf. on Quality Software*, 2001.

[35] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Symp. on Operating Systems Design and Implementation*, 2006.

A Transparently-Scalable Metadata Service for the Ursa Minor Storage System

Shafeeq Sinnamohideen[†] Raja R. Sambasivan[†] James Hendricks[†⋆]
Likun Liu[‡] Gregory R. Ganger[†]
[†]*Carnegie Mellon University* [‡]*Tsinghua University* [⋆]*Google*

Abstract

The metadata service of the Ursa Minor distributed storage system scales metadata throughput as metadata servers are added. While doing so, it correctly handles metadata operations that involve items served by different metadata servers, consistently and atomically updating the items. Unlike previous systems, it does so by reusing existing metadata migration functionality to avoid complex distributed transaction protocols. It also assigns item IDs to minimize the occurrence of multi-server operations. Ursa Minor's approach allows one to implement a desired feature with less complexity than alternative methods and with minimal performance penalty (under 1% in non-pathological cases).

1 Introduction

Ursa Minor is a scalable storage system designed to support automation [1]. It, like other direct-access storage systems [14], is structured in two primary parts: a data path for handling data access and a metadata path for handling metadata access. This separation allows each path to be optimized for its purpose. Modern scalable storage systems are expected to scale to thousands of storage nodes and tens or hundreds of metadata nodes, so each part is a distributed system in its own right. This paper explains the goals for the metadata path of Ursa Minor, describes the design and implementation of a prototype that fulfills them, and introduces and evaluates a novel technique for handling multi-server operations simply and efficiently.

As a whole, an Ursa Minor *constellation*, consisting of data nodes and metadata nodes, is expected to be highly available and durable, like other distributed storage systems. Ursa Minor is also intended to be incrementally scalable, allowing nodes to be added to or removed from the system as storage requirements change or hardware replacement becomes necessary. To provide for this, Ursa Minor must include a mechanism for *migrating* data from one data node to another and metadata from one metadata node to another.

The overall goals for Ursa Minor demand that the metadata path be *transparently scalable*. That is, it should be able to scale in capacity and throughput as more nodes are added, while users and client applications should not have to be aware of the actions the system takes to ensure scalability — the visible semantics should be consistent regardless of how the system chooses to distribute metadata across metadata nodes. Several existing systems have demonstrated this design goal (e.g., [3, 6, 36]).

The *scalable* part of the requirement implies that the system will use multiple metadata nodes, with each storing some subset of the metadata and servicing some subset of the metadata requests. In any system with multiple servers, it is possible for the load across servers to be unbalanced; therefore, some mechanism for load balancing is desired. This can be satisfied by migrating some of the metadata from an overloaded server to a less loaded one, thus relocating requests that pertain to that metadata.

The *transparent* part implies that clients see the same behavior regardless of how the system has distributed metadata across servers. Any operation on any single object should have the same result, no matter which server happens to be responsible for that object at that time. The same should hold for operations that involve more than one object, even if the objects involved are distributed to different servers, and even if some of those servers fail during the operation. This is an area that many existing systems leave unfinished for future work. Ursa Minor correctly and efficiently handles multi-server operations in a novel manner. The approach taken by Ursa Minor reuses the mechanism for migrating object metadata to implement multi-server operations by migrating the required metadata to a single metadata server and executing the operation on that server. This is a key contribution of our work because it simplifies the implementation of the metadata path while still providing good performance and the same failure semantics as a single server. To prevent most multi-file operations from being multi-server operations, Ursa Minor uses an object ID assignment policy that translates namespace locality into object ID locality.

Experimental results show that Ursa Minor scales linearly in metadata throughput when executing a variant of the SPECsfs97 benchmark. On workloads that contain frequent multi-server operations, performance degrades in proportion to the frequency of multi-server operations. This slowdown is due to the overhead of migration and is only 1.5% for workloads where multi-server operations are 10× more frequent than the worst-case behavior inferred from traces of deployed file systems.

The remainder of this paper is organized as follows: Section 2 reviews scalable distributed file systems, cross-server operations, and other related work. Section 3 describes the design of Ursa Minor's metadata path. Section 4 describes our evaluation of the prototype's performance and the trace analysis we performed in order to characterize expected workloads.

2 Background

Many distributed file systems have been proposed and implemented over the years. Architects usually aim to scale the capacity and throughput of their systems by doing one or more of the following:

- Increasing the capability of individual servers.
- Reducing the work each server performs per client.
- Increasing the number of servers in the system.

Each of these axes is largely independent of the others. As a research platform for exploring the issues of scaling distributed storage systems, Ursa Minor currently focuses on the last approach. Any work done here will still apply as the capability of each individual server improves. Such improvements would, of course, increase the capability of the entire constellation. Most existing work on decreasing a server's per-client workload focuses on the client-server protocols [18, 26, 32]. Historically, the adoption of improved protocols has been slowed by the need to modify every client system to use the new protocol. Recently, however, some of these techniques have been incorporated into the NFSv4 standard that is expected to be widely adopted [33]. Like the SpinFS protocol [11], Ursa Minor's internal protocol is designed to efficiently support the semantics needed by CIFS [27], AFS [18], NFSv4, and NFSv3 [8]. At present, however, we have only implemented the subset needed to support NFSv3.

As mentioned previously, challenges in scaling the number of servers in a system include handling the infrequent operations that involve multiple servers and managing the distribution of files across servers. The remainder of this section discusses operations that could involve multiple servers, how close existing systems come to being transparently scalable, how systems that handle multi-server operations transparently do so, and the importance of migration in a multi-server file system.

2.1 Multi-item operations

There are a variety of file system operations that manipulate multiple files, creating a consistency challenge when the files are not all on the same server. Naturally, every CREATE and DELETE involves two files: the parent directory and the file being created or deleted. Most systems, however, assign a file to the server that owns its parent directory. At some points in the namespace, of course, a directory must be assigned somewhere other than the home of its parent. Otherwise all metadata will be managed by a single metadata server. Therefore, the CREATE and DELETE of that directory will involve more than one server, but none of the other operations on it will do so. This section describes other significant sources of multi-item operations.

The most commonly noted multi-item operation is RENAME, which changes the name of a file. The new name can be in a different directory, which would make the RENAME operation involve both the source and destination parent directories. Also, a RENAME operation can involve additional files if the destination name exists (and thus should be deleted) or if the file being renamed is a directory (in which case, the '..' entry must be modified and the path between source and destination traversed to ensure a directory will not become a child of itself). Application programming is simplest when the RENAME operation is atomic, and both the POSIX and the NFSv3 specifications call for atomicity.

Many applications rely on this specified atomicity as a building-block to provide application-level guarantees. For example, many document editing programs implement atomic updates by writing the new document version into a temporary file and then using RENAME to move it to the user-assigned name. Similarly, many email systems write incoming messages to files in a temporary directory and then RENAME them into a user's mailbox directory. Without atomicity, applications and users can see strange intermediate states, such as two identical files (one with each name) existing or one file with both names as hard links.

Creation and deletion of hard links (LINK and UNLINK) are also multi-item operations in the same way that CREATE is. However, the directory the link is to be created in may not be the parent of the the file being linked to, making it more likely that the two are on different servers than for a CREATE and UNLINK.

The previous examples assume that each directory is indivisible. But a single heavily used directory might have more traffic than a single server can support. Some systems resolve this issue by splitting directories and assigning each part of the directory to a different server [37, 29]. In that case, simply listing the entire directory requires an operation on every server across which it is

split, and renaming a file within a directory might require two servers if the source name is in one part of the directory and the destination is in a different part.

Transactions are a very useful building block. Modern file systems, such as NTFS [28] and Reiser4 [31], are adding support for multi-request transactions. For example, an application could update a set of files atomically, rather than one at a time, and thereby preclude others seeing intermediate forms of the set. This is particularly useful for program installation and upgrade. The files involved in such a transaction could very easily be spread across servers.

Point-in-time snapshots [9, 17, 25, 30] have become a mandatory feature of most storage systems, as a tool for consistent backups, on-line integrity checking [25], and remote mirroring of data [30]. Snapshot is usually supported only for entire file system volumes, but some systems allow snapshots of particular subtrees of the directory hierarchy. In any case, it is clearly a substantial multi-item operation, with the expectation that the snapshot captures all covered files at a single point in time.

2.2 Transparent scalability

We categorize existing systems into three groups based on how fully they provide transparent scalability as the number of servers increases. Transparent scaling implies scaling without client applications having to be aware of how data is spread across servers; a distributed file system is not transparently scalable if client applications must be aware of capacity exhaustion of a single server or different semantics depending upon which servers hold accessed files.

No transparent scalability: Many distributed file systems, including those most widely deployed, do not scale transparently. NFS, CIFS, and AFS all have the property that file servers can be added, but each serves independent file systems (called *volumes*, in the case of AFS). A client can mount file systems from multiple file servers, but must cope with each server's limited capacity and the fact that multi-file operations (e.g., RENAME) are not atomic across servers.

Transparent data scalability: An increasingly popular design principle is to separate metadata management (e.g., directories, quotas, data locations) from data storage [6, 13, 14, 36, 38]. The latter can be transparently scaled relatively easily, assuming all multi-object operations are handled by the metadata servers, since each data access is independent of the others. Clients interact with the metadata server for metadata activity and to discover the locations of data. They then access data directly at the appropriate data servers. Metadata semantics and policy management stay with the metadata server, permitting simple, centralized solutions. The metadata server can limit throughput, of course, but off-loading data accesses pushes the overall system's limit much higher [14]. To go beyond this point, the metadata service must also be scalable.

Most modern storage systems designed to be scalable fall into this category. Most are implemented initially with a single metadata server, for simplicity. Examples include Google FS [13], NASD [14], Panasas [38], Lustre [24], prior versions of Ursa Minor [1], and most SAN file systems. These systems are frequently extended to support multiple metadata servers, each exporting a distinct portion of the namespace, and the ability to dynamically migrate files from one metadata server to another. Such a solution, however, is not transparently scalable because clients see different semantics for operations that cross metadata server boundaries.

Full transparent scalability: A few distributed file systems offer full transparent scalability, including Farsite [10], GPFS [32], Frangipani [36], and the version of Ursa Minor described in this paper. Most use the data scaling architecture above, separating data storage from metadata management. Then, they add protocols for handling metadata operations that span metadata servers. Section 2.3 discusses these further.

Another way to achieve transparent scalability is to use a virtualization appliance or "file switch" with a collection of independent NFS or CIFS file servers [7, 19, 20, 39]. The file switch aggregates an ensemble of file servers into a single virtual server by interposing on and redirecting client requests appropriately. In the case of multi-server operations, the file switch serves as a central point for serialized processing and consistency maintenance, much as a disk array controller does for a collection of disks. Thus, the virtual server remains a centralized, but much more capable, file system.

2.3 Multi-server operations

Traditionally, multi-server operations are implemented using a distributed transaction protocol, such as a two-phase commit [15]. Since each server already must implement atomic single-server operations, usually by using write-ahead logging and roll-back, the distributed transaction system can be built on top of the local transaction system. A transaction affecting multiple servers first selects one to act as a coordinator. The coordinator instructs each server to add a PREPARE entry, covering that server's updates, to their local logs. If all servers PREPARE successfully, the transaction is finalized with a COMMIT entry to all logs. If the PREPARE did not succeed on all servers, the coordinator instructs each server to roll back its state to the beginning of the transaction. With single-server transactions, recovering from a crash requires a server to examine its log and undo any incomplete transactions. Recovery from a multi-server transaction, however, is much more complicated.

With more than one server, it is possible for some servers to crash and others survive. If one crashed during PREPARE, the coordinator will wait until a time-out, then instruct the other servers to roll back their PREPAREs. If one crashed between PREPARE and COMMIT, when that server restarts, it needs to discover whether it missed the instruction to either COMMIT or UNDO. To do so, it needs to contact the coordinator or the other servers to determine whether any of them committed (in which case the coordinator must have successfully PREPAREd at all servers). Any step involving communication with other servers may fail, and if other servers have crashed, it may not be possible to proceed until they are online.

Distributed transactions may complicate other aspects of the system as well. Concurrency control within a single server requires each transaction to acquire locks to protect any state it operates on. The same is true for a multi-server operation, but now it is possible for the lock holder to crash independently of the server managing the lock. While there are existing techniques, such as leases, to handle this situation, a lock recovery scheme is simply not needed when locks can only be local to a server. Considering other common faults, such as an intermittent network failure, adds even more cases to handle.

As discussed, most of the additional complexity is in the recovery path. Not only must the recovery path handle recovery from a wide variety errors or crashes, it must also handle errors during recovery. This leads to a large number of cases that must be detected and handled correctly. Since errors in general are rare, and any particular error is even rarer, bugs in the fault-handling path may be triggered rarely and be even harder to reproduce. This places more reliance on test harnesses, which must be crafted to exercise each of the many error conditions and combinations thereof.

In order to minimize the rarely-used additional complexity of distributed transactions, Ursa Minor takes a novel approach to implementing multi-server operations. When a multi-server operation is required, the system migrates objects so that all of the objects involved in an operation are assigned to the same server, and the operation is then performed locally on that single server. This scheme is discussed in more detail in Section 3.5 and requires only single-server transactions and migration. Migration is itself a simplified distributed transaction, but it must already be implemented in the system to provide even non-transparent scalability.

2.4 Migration

In any system with many metadata servers, the question arises as to which files should be assigned to which servers. Some systems, such as AFS, NFS, Panasas, and Lustre, split the file system namespace into several *volumes* and assign each metadata server one or more volumes whose boundaries cannot be changed after creation. Others, such as xFS, Ceph, and OntapGX, are able to assign individual files to distinct servers. In general, supporting finer granularities requires more complexity in the mechanism that maps files to metadata servers.

Managing large-scale storage systems would be very difficult without migration — at the very least, hardware replacement and growth must be accounted for. Additionally, migration is a useful tool for addressing load or capacity imbalances. Almost every storage system has some way of performing migration, in the worst case by backing up data on the original server, deleting it, and restoring on the destination server.

Such offline migration, however, is obtrusive to clients, which will notice periods of data unavailability. If the need for migration is rare, it can be scheduled to happen during announced maintenance periods. As a system gets larger, the need for migration increases, while the tolerance for outages decreases. To address this issue, many modern systems [11, 18, 37, 38] can perform migration dynamically, while serving client requests, leaving clients unaffected except for very brief periods of unavailability. Any such system would be able to utilize the same approach used in Ursa Minor to provide transparent scalability.

The process of assigning files to servers can be thought of as analogous to lock management. A server that is assigned responsibility for a file (or collection of files) has effectively been granted an exclusive lock on that file and migration changes the ownership of that lock. Given the relative rarity and granularity of migration, the centralized migration managers used in AFS [18] and OntapGX [11] need not be as efficient or complex as the distributed lock managers used for fine-grained locking in GPFS [32], Slice [5], and Frangipani [36].

3 Design

Ursa Minor is a scalable storage system, designed to scale to thousands of storage nodes. Ursa Minor is a direct-access storage system [14], consisting of storage nodes and metadata servers. The storage nodes, termed *workers*, store byte streams named by Object ID, termed *SOID*[1]. There are no restrictions on which objects can reside on which worker, and an object's data can be replicated or erasure-coded across multiple workers, allowing the flexibility to tune an individual object's level of fault-tolerance and performance to its particular needs. Accessing a particular file's data requires two steps: first, the file name must be translated to a SOID, and second, the workers(s) responsible for the file data must be identified so that they can be contacted to retrieve the data. In Ursa Minor, these functions are performed by the Namespace Service (NSS) and the Metadata Service (MDS), respectively. This section describes the high-level organization of these services and provides more detail on the internal components that enable transparent scalability.

[1]A Self-* Object ID is a 128 bit number analogous to an inode number and unique across an Ursa Minor constellation

3.1 Metadata Service (MDS)

The Metadata Service in Ursa Minor maintains information on each object, similar to that maintained by the inodes of a local-disk file system. For each object, the MDS maintains a record that includes the object's size, link count, attributes, permissions, and the list of worker(s) storing its data. Clients communicate with the MDS via RPCs. Since clients are untrusted, the MDS must verify that each request will result in a valid state and that the client is permitted to perform that action. Some requests, such as creating or deleting an object, require the MDS to coordinate with workers. Others, such as updating an attribute or timestamp, reside wholly within the MDS. The semantics defined for the MDS imply that individual requests are atomic (they either complete or they don't), consistent (the metadata transitions from one consistent state to another), independent (simultaneous requests are equivalent to some sequential order), and durable (once completed, the operation's results will never be rolled back). The transaction mechanism used to ensure this is discussed in detail in Section 3.6.

The MDS is responsible for all object metadata in Ursa Minor. Individual object metadata records are stored in metadata *tables*. Each table includes all records within a defined range of SOIDs. The tables are internally structured as B-trees indexed by SOID and are stored as individual objects within Ursa Minor. The ranges can be altered dynamically, with a minimum size of one SOID, and a maximum of all possible SOIDS. Within those limits, the MDS may use any number of tables, and, collectively, the set of tables contains the metadata for all objects. Storing the tables as objects in Ursa Minor allows the MDS to benefit from the reliability and flexibility provided by Ursa Minor's data path, and results in the metadata path holding no hard system state.

Each Ursa Minor cluster includes one or more metadata servers. Each metadata server is assigned a number of metadata tables, and each table is assigned to at most one server at a time. Thus, accessing the metadata of any particular object will only involve one server at a time. Because the metadata tables are themselves objects, they can be accessed by any metadata server using Ursa Minor's normal data I/O facilities.

The assignment of tables to servers is recorded in a *Delegation Map* that is persistently maintained by a *Delegation Coordinator*. The delegation coordinator is colocated with one metadata server, termed the *Root Metadata Server*. This server is just like any other metadata server, except it happens to host the metadata for the objects used by the metadata service. Clients request the delegation map when they want to access an object for which they do not know which metadata server to contact. They cache the delegation map locally and invalidate their cached copy when following a stale cached delegation map results in contacting the wrong metadata server. Tables can be reassigned from one server to another dynamically by the delegation coordinator, and this process is discussed in Section 3.4.

3.2 Namespace Service (NSS)

The Namespace Service manages directory contents. Directories are optional in Ursa Minor — applications satisfied with the MDS's flat SOID namespace (e.g.: databases, mail servers, scientific applications) need not use directories at all. Other applications expect a traditional hierarchical directory tree, which the Namespace Service provides.

Similarly to a local-disk file system, a directory entry is a record that maps a filename to a SOID. Directories are B-tree structured, indexed by name, and stored as ordinary objects, with their own SOIDs. At present, each directory object contains all of the directory entries for that directory, though there is no obstacle to splitting a directory across multiple objects.

Namespace servers are tightly coupled with metadata servers (in our implementation, both are combined in one server process which exports both RPC interfaces). Each namespace server is responsible for directories whose SOIDs are within the range exported by its coupled metadata server. This ensures that a directory's "inode" (the attributes stored by the MDS) and its contents will always be served by the same process. For the rest of this paper, we use the term "metadata server" to refer to the combined MDS and NSS server.

The NSS aims to support directory semantics sufficient to implement an overlying file system with POSIX, NFS, CIFS, or AFS semantics. As such, it provides the POSIX notions of hard links, including decoupling of unlink and deletion, and the ability to select how already-existing names are handled. Typical operations include creating a file with a given name, linking an existing object under a new name, unlinking an file, looking up the SOID corresponding to a file name, and enumerating the contents of directories.

3.3 SOID assignment

In Ursa Minor, the SOID of an object determines which table, and thus which metadata server, that object is assigned to. It follows that there may be advantages in choosing to use particular SOIDs for particular files. For instance, the **ls -al** command will result in a series of requests, in sequential order, for the attributes of every file in a given directory. If those files all had numerically similar SOIDs, their metadata would reside in the same (or nearby) B-tree pages, making efficient use of the server's page cache. Similarly, most file systems exhibit spatial locality, so an access to a file in one directory means an access to another file in that same directory is likely. Secondly, many directory operations (CREATE, LINK) operate on both a parent directory and a child inode at the same time. If the parent and child had nearby SOID numbers, they would likely reside in the same table, simplifying the transaction as discussed in Section 3.5 and Section 3.6.

For these reasons, it would be useful to assign SOIDs such that children of a directory receive SOIDs similar to those of the directory itself. Applied over a whole directory tree, a *namespace flattening policy* would convert "closeness" in the directory hierarchy to "closeness" in SOID values. A number of algorithms could be used for this task; we use a *child-closest policy* [16], which works as follows.

First, the SOID is divided bitwise into a *directory segment* and a *file segment*. The directory segment is further subdivided into a number of *directory slots*. Each slot corresponds to a level in the directory hierarchy, and the value in a slot identifies that directory within its parent. The root directory uses the most significant slot, each of its children the next most significant, and so on. When creating a new directory, the child's directory segment is copied from its parent, with a new value chosen for the most significant empty directory slot.

The file segment is simple sequential counter for files created in that directory. A directory itself has a file segment of all 0s. The first child file of that directory has the same directory segment, but file segment of 1. The second has file segment of 2 and so on. Figure 1 shows an example directory tree and the SOID the child-closest policy assigns to each file or directory in the tree.

This scheme is similar to that used in Farsite, except that the Farsite FileID is variable-length and grows with with directory depth [10]. Supporting variable-length object identifiers would unduly complicate the implementation of Ursa Minor's protocols and components, so we use a fixed-size SOID.

With a fixed-size SOID, the namespace may have both more levels than there are directory slots and more files in a directory than can be represented in the file segment bits. To accommodate this, 2 prefix bits are used to further split the SOID into 4 regions. The first, *primary*, region, uses the assignment policy above. If the hierarchy grows too deep, the too-deep child directory is assigned a new top-level directory slot with a different prefix (the *too-deep* region). Its children grow downwards from there, as before.

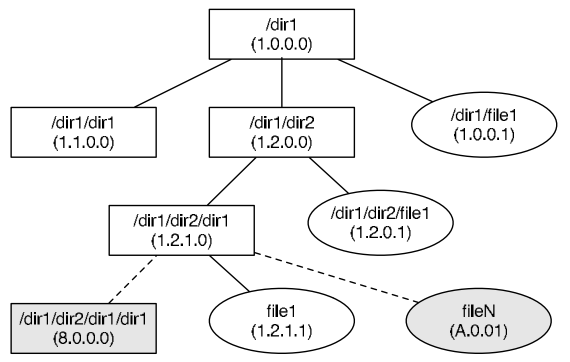


Figure 1: Child-closest SOID assignment policy. The SOID chosen for each element of this simple directory tree is shown. For clarity the example uses a 16 bit SOID and a "." is used to separate the value of each 4 bit directory slot and file segment. The dashed lines show a too-deep directory overflowing to a new "root" and a file in a large directory overflowing to the *too-wide* region.

If there are too many files in a directory and the next directory slot value is unused, the large directory takes over the SOIDs reserved for its nonexistent sibling and the new file is assigned a SOID that would used by its nonexistent cousin. If cousins already exist, the new file is assigned a SOID from the *too-wide* region. Within this region, fewer bits are allocated to the directory segment, and more to the file segment, so more files per directory can be handled. Finally, if either of these additional regions overflow, the *catch-all* prefix is used, and SOIDs are assigned sequentially from this region.

In the case of any overflow, the additional children are effectively created under new "roots" and thus have very different SOIDs from their parents. However, those children will still have locality with their own children (the parent's grandchildren). Thus, one large subtree will be split into two widely separated subtrees, each with locality within itself. If the two subtrees are both large enough, the loss of locality at the boundary between subtrees should not have a significant effect because most operations will be local to one subtree or the other.

By tuning the bit widths of the directory segment, file segment, and directory slots to match the system's workload, instances of overflow can be made extremely rare [16]. Namespace manipulations, such as linking or renaming files, however, will result in the renamed file having a SOID that is not similar to the SOID of its new parent or siblings. The similar situation happens in local disk files systems: a renamed file's inode still resides in its original cylinder group after a rename.

The SOID of a deleted file is available for re-use as soon as the file's storage has been reclaimed from the relevant workers (this step is performed lazily in most cases). Thus, as long as a directory's size does not

change over time, changing its contents does not affect the chance of overflow. In fact, reusing a SOID as soon as possible should provide for a slight efficiency gain, by keeping the metadata B-tree compact.

In all of these cases, outside of the SOID selection policy, MDS treats the SOID as an opaque integer and will operate correctly regardless of how much or little locality the SOIDs preserve. Performance will be be better with higher locality, however. The segment sizes do not need to remain constant over the life of a constellation, or even across the SOID namespace, so there is the potential to adaptively tune them based on the observed workloads, however we have not yet implemented this.

The net effect of combining namespace flattening with SOID-range tables is that each table usually ends up containing a subtree. This is somewhat analogous to the volume abstraction offered by systems like AFS but without the predefined, rigid mapping of subtree to volume. Unlike these systems, a too-large or too-deep subtree will overflow into another table, quite possibly not one served by the same server. One can think of these overflowed subtrees as being split off into separate sub-volumes, as is done in Ontap GX and Ceph.

3.4 Metadata migration

Ursa Minor includes the ability to dynamically migrate objects from one metadata server to another. It does so by reassigning responsibility for a metadata table from one server to another. Because the metadata table (and associated directories) are Ursa Minor data objects accessible to all metadata servers, the contents of the metadata table never need to be copied. The responsibility for serving it is simply transferred to a different server. This section describes the process for doing so in more detail.

Each metadata server exposes an RPC interface via which the delegation coordinator can instruct it to ADD or DROP a table. In order to migrate table T from server A to server B, the coordinator first instructs server A to DROP responsibility for the table. When that is complete, the coordinator updates the delegation map to state that B is responsible and instructs server B to ADD T. At all times, at most one server is responsible.

When server A is instructed to DROP T, it may be in the process of executing operations that use T. Those operations will be allowed to complete. Operations waiting for T will be aborted with an error code of "wrong server", as will any new requests that arrive. Clients that receive such a response will contact the coordinator for a new delegation map. Once the table is idle, server A sets a bit in the table header to indicate the table was cleanly shut down, flushes the table from its in-memory cache, and responds to the coordinator that the table has been dropped.

Adding a table to server B is also simple. When instructed to ADD responsibility, server B first reads the header page of table T. Since T's header page indicates it was shut down cleanly, no recovery or consistency check procedure is necessary, so server B simply adds an entry for T to its in-memory mapping of SOID to table. Any subsequent client requests for SOIDs within T will fault in the appropriate pages of T. Before its first write to T, server B will clear the "clean" bit in the header, so any subsequent crash will cause the recovery procedure to run.

3.5 Multi-object operations

For a server, performing a transaction on a single object is simple: acquire a local lock on the SOID in question and on the SOID's table, perform the operation, and then release all locks.

Performing a transaction with multiple objects or tables within a single server is similar, but complicated by the need to avoid deadlocks between operations that try to acquire the same locks in opposite orders. Each server's local lock manager avoids deadlock by tracking all locks that are desired or in use. When all locks required for an operation are available, the lock manager acquires all of them simultaneously and allows the operation to proceed.

In the more complicated case (shown in Figure 2) of a multi-object and multi-server operation, the server's local lock manager will discover that all the required resources are not local to the server. The lock manager blocks the operation and sets out to acquire responsibility for the required additional tables. To do so, it sends a BORROW request to the Delegation Coordinator. The BORROW request includes the complete list of tables required by the operation; the coordinator's lock manager will serialize conflicting BORROWs. When none of the tables required by a BORROW request are in conflict, the coordinator issues a series of ADD and DROP requests to move all the required tables to the requesting server and returns control to it. Those tables will not be moved again while as the transaction is executing.

When the transaction completes, the requesting server sends a RETURN message to the coordinator, indicating it no longer requires exclusive access to that combination of tables. The coordinator determines whether it can now satisfy any other pending BORROW requests. If so, the coordinator will migrate a RETURNed table directly to the next server that needs it. Otherwise the coordinator can choose to either migrate that table back to its original server (the default action), leave it in place until it is BORROWed in the future, or migrate it to some other server. Note that, while waiting for a BORROW, a server can continue executing other operations on any ta-

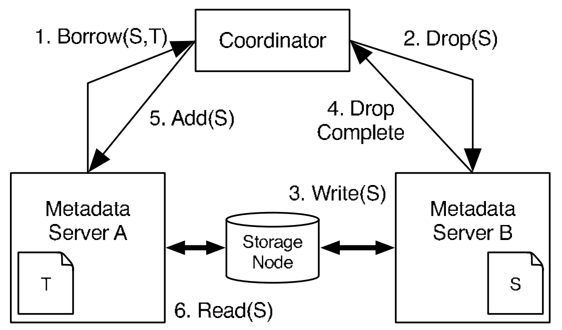


Figure 2: Borrowing a table The sequence of operations required for Server A to handle an operation requiring tables S and T, when table T is initially assigned to server A and table S to server B. Returning to the original state is similar.

bles it already has; only the operation that required the BORROW is delayed.

3.6 Transactions

Underlying the Metadata and Namespace Services is a transactional layer that manages updates to the B-tree structures used for storing inodes and directories. These B-trees are stored as data in Ursa Minor objects. The data storage nodes and their access protocols guarantee that individual B-tree pages are written atomically to the storage nodes and that data accepted by the storage nodes will be stored durably. The transaction system extends these guarantees to transactions involving multiple B-tree pages spread across multiple B-trees.

Atomicity is provided using a simple shadow-paging scheme. All updates to the B-tree data object are deferred until commit time. The data object includes two storage locations for each page, and the location written alternates on every write of that page. Thus, one location will contain the most recent version of that page, and the other location will contain the next most recent version. Each page includes a header that links it to all the other pages written in the same transaction, which will be used by the recovery mechanism to determine whether the transaction committed or needs to be rolled back. Reading a page requires reading both locations and examining both headers to identify the latest version. The server may cache this information, so subsequent rereads only need the location with the latest page contents.

Isolation is guaranteed by allowing only a single transaction to execute and commit on each B-tree at a time. Every transaction must specify, when it begins, the set of B-trees it will operate on. It acquires locks for all of those B-trees from the local lock manager, and holds them until it either commits or aborts. If, during execution, the transaction discovers it needs to operate on a B-tree it does not hold a lock for, it aborts and restarts with the new B-tree added to the set. This strategy is similar to that used by Sinfonia mini-transactions, which share the limitation of specifying their read and write sets up front [4]. Most transactions require only a single execution. The main sources of repeated executions are operations that traverse a file system path: at each step, the SOID of the next directory to read is determined by reading the current directory.

Consistency is only enforced for the key field of the B-tree records; maintaining the consistency of the data fields is the responsibility of the higher level code that modifies them.

Durability is provided by synchronously writing all modified pages to the storage nodes at commit time. The storage nodes may either have battery-backed RAM or themselves synchronously write to their internal disk.

If the metadata server crashes while committing a transaction, it is possible for the B-tree to be in an inconsistent state: for example, only 2 of the 3 pages in the last transaction may have been written to the storage nodes before the crash. To resolve this condition, the metadata server performs a recovery process when it restarts after an unclean shutdown. First, it queries each storage node to determine the location of the last write to the B-tree object (the storage node must maintain this information as part of the PASIS protocol [2]). The location of the last write corresponds to the last page written. Reading that page's header will reveal the identity of all other pages that were part of the same transaction. If all the other pages have transaction numbers that match that of the last written page, then we know that the transaction completed successfully. If any of them has an earlier transaction number, we know that not all page writes were completed, and a rollback phase is performed: any page with the latest transaction number is marked invalid, and its alternate location is marked as the valid one. At the end of rollback, the latest valid version of every page is the same as it was before the start of the rolled-back transaction. The recovery process can proceed in parallel for B-trees with independent updates, whereas two B-trees involved in the same transaction must be recovered together. Because there is at most one transaction committing at a time on a given B-tree, at most one rollback on a given B-tree will be necessary.

3.7 Handling failures

Any of the large number of components of the metadata path can fail at any time, but all failures should be handled quickly and without data loss. In general, our design philosophy considers servers trustworthy; we are primarily concerned with crashes or permanent failure and not with faulty computations or malicious servers.

The most obvious components to consider for failure are the metadata server software and the hardware that

it runs on. A constellation monitoring component polls all metadata servers (as well as other components) periodically, and if the server does not respond within a time-out interval, that metadata server instance is considered to have failed. The monitoring component will then attempt to start a replacement metadata server instance, either on the same hardware or on a different node. The new instance queries the delegation coordinator to determine the tables for which it is responsible and runs the recovery process. After recovery completes, the new instance is in exactly the same state as the previous instance. While the new instance is starting and recovering, client requests sent to the old instance will time-out and be retried.

It is possible that, due to a network partition, a properly operating metadata server may be incorrectly declared by the system monitor to have failed. Restarting a new instance would result in two servers trying to serve the same objects, violating the consistency assumptions. To avoid this, the delegation coordinator revokes the capabilities used by the old instance to access its storage nodes before granting capabilities to the new instance to do the same. Thus, while the old instance may still be running, it will not be able to access its backing store, preserving consistency; nor will clients be able to use capabilities granted by old the instance to access client data. If the revocation attempt fails to reach a quorum of storage nodes, perhaps because they are also on the other side of the network partition, the coordinator will not start a new server instance until the partition heals and the old instance continues uninterrupted until then.

Not only does a failed metadata server affect clients, but it may also affect another server if it failed in the middle of a migration. The delegation coordinator will see its ADD or DROP request time out and propagate this error to any operation that depended on the migration. The metadata being migrated will be unavailable until the metadata server restarts, just like any other metadata served by the failed server. It is reasonable for a multi-server operation to fail because one of the servers it needs is unavailable.

When the failed metadata server restarts, the delegation map it receives from the coordinator will be unchanged from when the server began its last ADD or DROP: a failed ADD will be completed at this time, and failed a DROP effectively never happened. Instead of waiting for a server to restart, the tables assigned to the failed server could simply be reassigned to other working servers. Doing so, however, complicates the process of recovering a table that was involved in a multi-table (but same server) transaction: As described in Section 3.6, both tables must be recovered together, which poses a problem if the two tables have been reassigned to different servers for recovery. Although it is possible to detect and handle this case, in the interest of simplicity, we avoid it by always trying to recover all the tables assigned to a failed server as one unit.

A failed delegation coordinator will prevent the system from performing any more delegation changes, although all metadata servers and clients will continue to operate. As the delegation map is stored in an object and synchronously updated by the coordinator, the coordinator is stateless and can simply be restarted the same as metadata servers. There must be at most one delegation coordinator in a constellation. One method to ensure this, that we have not yet implemented, is to use a quorum protocol to elect a new coordinator [21].

If the failure happened during a migration, the metadata table(s) being migrated will be in one of two states: the delegation map says server A is responsible for table T but server A does not think it is, or the delegation map says no server is responsible for T. The delegation map is always updated in an order such that a server will never be responsible for a metadata table that is not recorded in the delegation map. To handle the first case, a newly started coordinator will contact all metadata servers to determine which tables they are serving and issue the appropriate ADD requests to make the server state match the delegation map. In the second case, an appropriate server is chosen for tables that have no assigned server, and an ADD request is issued.

For storage node failures, we rely on the Ursa Minor data storage protocol to provide fault tolerance by replicating or erasure-coding object data across multiple storage nodes. Since the contents of the metadata tables cannot be reconstructed from any other source, they must be configured with appropriately high fault tolerance.

4 Evaluation

Our goal was to construct a transparently scalable Metadata Service for Ursa Minor. To show we have succeeded, we evaluate the performance of Ursa Minor with a standard benchmark as well as with a range of modified workloads to reveal its sensitivity to workload characteristics. Section 4.1 describes the benchmark's workload, Section 4.2 describes the hardware and software configurations used, Sections 4.3 and 4.4 discuss experimental results, Section 4.5 contrasts these results with the workloads seen in traces of deployed file systems, and Section 4.6 discusses additional observations.

4.1 Benchmark

The SPECsfs97 [35] benchmark is widely used for comparing the performance of NFS servers. It is based on a survey of workloads seen by the typical NFS server and consists of a number of client threads, each of which emits NFS requests for file and directory operations according to an internal access probability model. Each thread creates its own subdirectory and operates entirely within it. Since each thread accesses a set of files independent from all other threads, and each thread only has a single outstanding operation, this workload is highly parallelizable and contention-free.

In fact, using the namespace flattening policy described in Section 3.3, Ursa Minor is trivially able to assign each thread's files to a distinct SOID range. Thus, each metadata table consists of all the files belonging to a number of client threads, and all multi-object operations will only involve objects in the same table. While this is very good for capturing spatial locality, it means that multi-server operations will never occur for the default SPECsfs97 workload.

Because the SPECsfs97 benchmark directly emits NFS requests, these requests must be translated into the Ursa Minor protocol by an *NFS head-end*. Each head-end is an NFS server and an Ursa Minor client, and it issues a sequence of Ursa Minor metadata and/or data requests in order to satisfy each NFS request it receives. In the default SPECsfs97 workload, 73% of NFS requests will result in one or more Ursa Minor metadata operations, and the remaining 27% are NFS data requests that may also require an Ursa Minor metadata operation. Like any Ursa Minor client, the head-end can cache metadata, so some metadata operations can be served from the head-end's client-side metadata cache, resulting in a lower rate of outgoing Ursa Minor metadata requests than incoming NFS requests. Each head-end is allocated a distinct range of SOIDs for its use, and it exports a single NFS filesystem. Thus, different head-ends will never contend for the same objects, but the client threads connected to a head-end may access distinct objects that happen to be in the same metadata table.

In order to use SPECsfs97 to benchmark Ursa Minor, we found it necessary to make a number of practical modifications to the benchmark parameters and methodology specified by SPEC. First, we modified the configuration file format to allow specifying operation percentages in floating point as necessary for Section 4.4. Second, we doubled the warmup time for each run to 10 minutes to ensure the measured portion of the run did not benefit from startup effects. Neither of these changes should affect the workload presented during the timed portion of the run.

Because we are interested in exploring the scalability of the MDS, we must provision the Ursa Minor constellation so that the MDS is always the bottleneck. Doing so requires enough storage nodes to collectively hold the metadata objects in their caches — otherwise, the storage nodes become the bottleneck. The number of files used by SPECsfs97 is a function of the target throughput and, at high load levels, would require more storage nodes than we have available. Additionally, as the number of files varies, so will the miss-rate of the fixed-size head-end caches, changing the workload seen by the MDS. To avoid these two effects, we use a constant 8 million or 4 million files, requiring 26 GB or 13 GB of metadata. To avoid confusion, we refer to this modified benchmark as *SFS-fixed*.

To maximize MDS load, we configured the NFS head-ends to discard any file data written to them and to substitute zeroes for any file data reads. The Ursa Minor metadata operations associated with the file read and writes are still performed, but the Ursa Minor data operations are not, so we can omit storage nodes for holding file data. In all other regards, including uniform access, we comply with the SPECsfs97 run reporting rules.

4.2 Experimental setup

Table 1(a) lists the hardware used for all experiments, and Table 1(b) lists the assignment of Ursa Minor components to physical machines. This particular assignment was chosen to ensure as uniform hardware and access paths as possible for each instance of a component — every storage node was the same number of network hops away from each metadata server, and each head-end was the same distance from each metadata server.

We configured the test constellations with the goal of ensuring that the MDS was always the bottleneck. The root metadata server was only responsible for objects internal to the MDS (i.e., the metadata for the metadata table objects themselves). The large constellation had 48 NFS head-ends, each serving 20 SFS-fixed client threads (960 in total), and the SOID range assigned to each head-end was split across 8 tables. The resulting 384 tables were assigned in round-robin fashion across metadata servers, such that every head-end used some object on each metadata server. Similarly, tables were stored on 24 storage nodes such that each metadata server used every storage node. These choices increase the likelihood of multi-table operations and contention and are intended to be pessimistic. For small experiments, we used 24 head-ends, 12 storage nodes, and 480 client threads. The SOID assignment policy was configured to support a maximum of 4095 files per directory and 1023 subdirectories per directory, which was sufficient to avoid overflow in all cases. Each storage node used 1.6 GB of battery-backed memory as cache. In addition, 256 MB at each metadata server was used for caching B-tree pages, and the head-ends had 256MB each for their client-side caches.

4.3 Scalability

Figure 3 shows that the Ursa Minor MDS is transparently scalable for the SFS-fixed workload. Specifically, the throughput of the system for both NFS and MDS operations increases linearly as the number of metadata servers increases. This is as expected, because the basic SPECsfs97 and SFS-fixed workloads cause no multi-server operations. Thus, adding additional servers evenly divides the total load across servers. Because the head-end servers include caches and because the SFS operation rate includes NFS data requests, the number of requests that reach the metadata servers is lower than that seen by the head-end servers. However, the workload presented to the MDS is much more write-heavy — 26% of requests received by the MDS modify metadata, compared the 7% of NFS requests that definitely will modify metadata and 9% that possibly will.

Type	Type A	Type B
Count	38	75
RAM	2 GB	1 GB
CPU	3.0 Ghz Xeon	2.8 Ghz Pentium 4
Disk	4× ST3250823AS	1× WD800J
NIC	Intel Pro/1000 MT	Intel Pro/1000 XT
OS	Linux 2.6.26	
Switch	3× HP ProCurve 2848	

(a) Hardware configuration.

Component	Hardware	Large	Small
Storage nodes	Type A	24	12
Metadata servers	Type B	8-32	4-16
NFS head-ends	Type B	48	24
Load generators	Type A	5	2
Root metadata server Root storage node Constellation manager	Type A	1	1

(b) Ursa Minor configuration.

Table 1: Hardware and software configuration used for large and small experiments. The number of metadata servers used varied; all other components remained constant. The root metadata server and its storage node only stored metadata for objects used by the metadata service. Metadata accessible by clients was spread across the remaining storage nodes and metadata servers.

4.4 Multi-server operations

Since Ursa Minor uses a novel method of implementing multi-server operations, it is important to consider its performance on workloads that are are less trivially parallelizable. To do, so we modified the base SFS-fixed workload to include a specified fraction of cross-directory LINK operations. To keep the total number of operations constant, we reduce the number of CREATE operations by one for every LINK operation we add. Thus, the sum of LINK and CREATE is a constant 1% of the NFS workload. The resulting MDS workload contains a higher fraction of both, because the head-end cache absorbs many of the LOOKUP requests.

We also modify the namespace flattening policy so each client thread's directories are spread across all the tables used by that head-end, giving a $1-(1/N)$ chance any LINK being multi-server. Both LINK and CREATE modify one directory and one inode, so any performance difference between the two can be attributed to the overhead of performing a multi-server operation. A RENAME, however, modifies two directories and their inodes, and is slower than a CREATE even on a single server, which is why we use LINK as the source of cross-directory operations in this experiment.

Figure 4 shows the reduction in MDS throughput for SFS-fixed with multi-server operations compared to SFS-fixed without multi-server operations. Workloads in which 0.05% to 1.00% of NFS operations were cross-directory LINKs resulted 0.07% to 4.75% of MDS operations involving multiple servers. Separate curves are shown for Ursa Minor configurations where each head-end's metadata is split across 16, 8, or 4 tables for a total of 384, 192, or 96 tables in the system. As expected, throughput decreases as the percentage of multi-server ops increases, since each multi-server op requires a table migration. Accordingly, the latency of LINKs are up to $3.5\times$ that of CREATEs (120 ms vs. 35 ms). Furthermore, when the table is RETURNed to its original server, that server's cache will not contain any of the migrated table's contents. The resulting increase in cache miss rate decreases the throughput of subsequent single-server operations [34].

Additionally, migrating a table makes it unavailable for serving other operations while the migration is in progress. When a single table represents a small fraction of the total metadata in the system, making one table unavailable has a small impact on overall performance. However, if we configure the system to fit the same metadata into fewer tables, the penalty increases, as shown in Figure 4. This is exacerbated in Ursa Minor because threads within the metadata server frequently contend for table-level locks in addition to CPU and I/O. Given that small tables permit finer-grained load balancing, a reasonable Ursa Minor configuration might place 1%-10% of a server's capacity in a single table as suggested for other systems [6]. The major penalty of having far too many tables is that the delegation map will be large, possibly requiring a more efficient means of storing and distributing it.

4.5 Trace analysis

To put the performance of Ursa Minor under multi-server operations into context, we examined two sets of well-studied distributed file system traces to determine what rates of multi-server operations are seen in real-world workloads. Table 2 classifies the operations in each trace

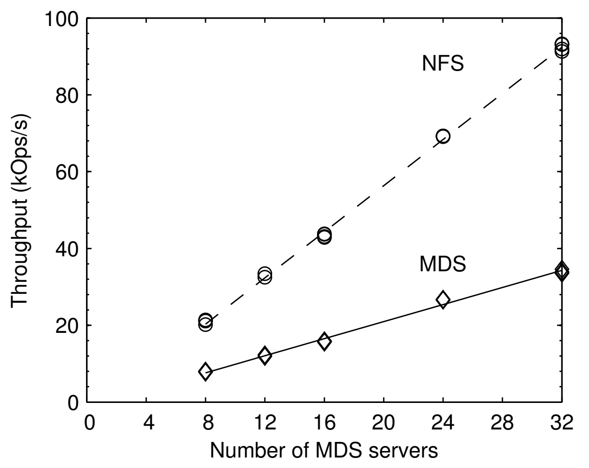


Figure 3: Throughput vs. number of metadata servers. The number of NFS and MDS operations completed per second are shown separately for the SFS-fixed workload. The difference between NFS and MDS operation rates is due to data-only requests and the head-end's metadata cache. The three benchmark runs performed for each configuration are plotted individually. The lines show a linear fit with correlation coefficient of > 0.995. All runs used 8 million files on the large constellation. Our present implementation is limited to about 1 million files per server; plots with fewer files and servers are similar [34].

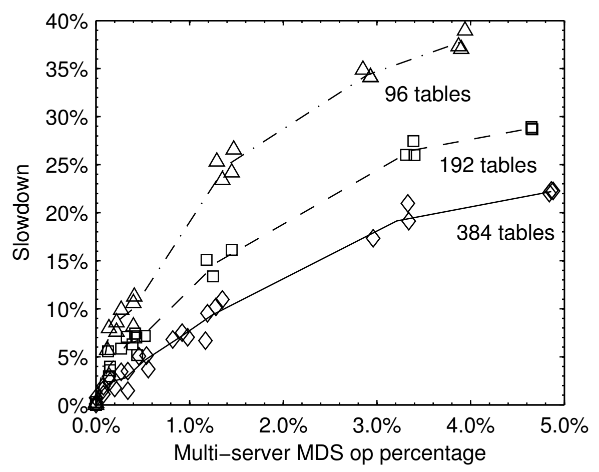


Figure 4: Slowdown vs. percentage of multi-server ops. The slowdown in MDS throughput (compared to a workload with no multi-server ops) is shown for SFS-fixed workloads with varying percentages of multi-server LINKs. The actual percentage achieved in a given run varies from the target percentage; the actual percentage is plotted for each run, and the lines connect the average of all runs with the same target percentage. All runs use 4 million files on the small constellation with 12 metadata servers. The solid line uses the same configuration of 384 total tables used in Figure 3, additional lines use 192 and 96 tables.

by the Ursa Minor metadata operation that would be required to service it. While all operations except for UPDATEs and some LOOKUPs involve more than one object, those objects are almost always in a parent-child relationship. In any system that preserves namespace locality (as the child-closest SOID assignment policy in Ursa Minor does), both objects will be served by the same metadata server. The exceptions are operations on mountpoints, operations on directories that are extremely large, and operations that involve more than one directory. Since the first case should be extremely rare, we expect that cross-directory operations will be the major source of multi-server metadata operations.

The first set of traces are of 3 departmental NFS servers at Harvard University. The workload of each server varied significantly and is described by Ellard et al. [12]. In these traces, RENAME operations may involve more than one directory, and we count cross-directory RENAMEs separately from RENAMEs of a file to a different name in the same directory. Additionally, a LINK operation, while only involving a single file and single directory, might be adding a link in one directory to a file originally created in a different directory. While the original directory does not matter to a traditional NFS server, in Ursa Minor, the file's SOID will be similar to that of its original parent directory, while the new parent directory may have a very different SOID and perhaps be on a different server. Unfortunately, unlike RENAMEs, the LINK RPC does not contain enough information to reliably identify the original parent directory, so we cannot separate these cases. The highest fraction of cross-directory RENAME operations occurred in the DEAS trace and represented only 0.005% of operations in that trace.

The second set of traces are of CIFS traffic to two enterprise-class filers in NetApp's corporate data center. One was used by their engineering department, the other by the their marketing, sales, and finance departments. The fraction of RENAME operations in these traces is similar to those from Harvard, though the distribution of other operations is very different. In the CIFS protocol, the equivalent of the RENAME RPC includes the path of the source and destination directories, so it is possible to determine not only that the the directories are different, but how far apart the source and destination directories are in the directory tree. We were able to analyze a segment of the trace from the Corporate server to calculate rename distances for operations within that segment. Of the 80 cross-directory RENAMEs we found, 56% had a destination directory that was either the immediate parent or child of the source directory.

For comparison, we also show the distribution of operations in the default configuration of SPECsfs97 in Table 2. In all of these workloads, the percentage of cross-directory operations is very low. And, of those cross-directory operations, only a fraction will be multi-server. If directories were assigned randomly to servers, the probability both directories will happen to be on the same server is $1/N$. If the directories involved exhibit spatial locality, as the CIFS traces do, and the OID as-

	EECS	DEAS	CAMPUS	Engineering	Corporate		SPECsfs97
Total operations	180M NFS	770M NFS	672M NFS	352M CIFS	228M CIFS	12.5M CIFS	4.9M NFS
LOOKUP	93.113%	98.621%	97.392%	87.1%	73.2%	62.417%	83.000%
CREATE	0.772%	0.243%	0.286%	0.7%	6.7%	13.160%	1.000%
DELETE	0.810%	0.431%	0.555%	0.006%	0.03%	0.030%	1.000%
UPDATE	5.250%	0.683%	1.766%	1.24%	2.2%	14.282%	14.606%
RENAME (all)	0.054%	0.022%	$< 0.001\%$	0.02%	0.04%	0.036%	0.000%
RENAME (cross-dir)	0.0012%	0.005%	$< 0.001\%$	NA	NA	$< 0.001\%$	0.000%

Table 2: Metadata operation breakdowns for various distributed filesystem traces. The percentage of operations in the original trace that incur each type of Ursa Minor metadata operation is shown. This represents the workload that would seen by the Ursa Minor head-end's metadata cache. Only LOOKUP requests are cacheable, thus we expect the workload seen by the metadata servers to have fewer LOOKUPS. The columns do not sum to 100% because of not all CIFS or NFS operations require Ursa Minor metadata. For the large CIFS traces, the values are calculated from CIFS operation statistics provided by Leung et al. [23] and represent an upper bound for each operation. For the NFS trace and the small CIFS trace, we scan the trace and count the resulting operations. The operations generated by a 5 minute run of SPECsfs97 at 16000 ops/sec are shown for comparison. In all workloads, RENAMES that involve two directories are shown separately and are extremely rare.

signment policy can preserve spatial locality, then both directories are far more likely to be on the same server. Even pessimistically assuming that all cross-directory operations are multi-server, Ursa Minor's approach to multi-server operations can handle an order of magnitude more multi-server operations (.06%) with only a 1.5% decrease in overall throughput compared to a workload with only single-server operations. A system that could execute multi-server operations as fast as single-server ones would be optimal. Even if the workload contains 1% multi-server operations, the slowdown is 7.5%, but such a high rate seems unrealistic, given the rarity of even potentially multi-server operations.

4.6 Additional observations

Our motivation for using migration to handle multi-server operations was that it was the simple solution for the problem at hand. From the starting point of a metadata service that supported migration and single-server operations (over 47000 lines of C code), it only required 820 additional lines of code to support multi-server operations. Of these 820 lines, the global lock manager (necessary for avoiding deadlock) accounted for 530 lines, while the remainder were additional RPC handlers and modifications to the local transaction layer to trigger a BORROW when necessary. In contrast, implementing migration correctly represented 9000 lines of the original metadata server and several months of work.

To provide a basis for comparison, we created a version of Ursa Minor that implements multi-server operations using the traditional 2-phase commit protocol. This version is not nearly as robust or stable as the main version, particularly with regard to handling and recovering from failures, so the 2587 lines required to implement it represent a lower bound. The code to implement a write-ahead log is not included in this total because most other systems include one as part of their basic functionality.

Many of the choices we made in designing the MDS were guided by the properties of the rest of Ursa Minor. Other systems with different underlying storage or failure models might choose to store metadata on the local disks or NVRAM of each metadata server. Migration in such a system would be much more expensive because it requires copying metadata from server to server.

The single delegation coordinator is involved in every multi-server operation, and could become a bottleneck as the constellation scales. We found the coordinator was capable of up to 3500 migrations per second, which is reached with 32 metadata servers and a workload with 1% multi-server ops. Scaling beyond this point would require moving to a hierarchy of coordinators rather than a single one. More details, along with discussion of other workloads and system parameters, are presented in an additional technical report [34].

5 Conclusion

Transparent scalability for metadata is a desirable feature in a large storage system. Unfortunately, it is a difficult feature to provide because it introduces the possibility of multi-server operations, which in turn require relatively complex distributed protocols. By reusing metadata migration to reduce multi-server operations to single-server ones, we were able to implement a transparently scalable metadata service for Ursa Minor with only 820 additional lines of code. Although this approach is more heavyweight than a dedicated cross-server update protocol, the performance penalty is negligible if cross-server operations are are as rare as trace analysis suggests — less than 0.005% of client requests could possibly be cross-server in the traces analyzed. Even if all of those requests were in fact cross-server, Ursa Minor can tolerate an order of magnitude more cross-server operations (.06%) with only a 1.5% decrease in overall throughput. We believe that this approach to handling infrequent cross-server operations is very promising for distributed file systems and, perhaps, for other scalable distributed systems as well.

Acknowledgements

We thank our shepherd, Stephen Hand, and the reviewers for their insightful comments. We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard, Hitachi, IBM, Intel, LSI, Microsoft, NEC, NetApp, Oracle, Seagate, Symantec, VMware, and Yahoo!) for their interest, insights, feedback, and support. We also thank Intel, IBM, NetApp, Seagate and Sun for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620–01–1–0433, and by CyLab at Carnegie Mellon University under grant DAAD19–02–1–0389 from the Army Research Office. James Hendricks was supported in part by an NDSEG Fellowship, which is sponsored by the Department of Defense. Likun Liu is supported by the Natural Science Foundation of China via project #60963005 and the National Basic Research (973) Program of China via project #2007CB310900.

References

[1] M. Abd-El-Malek, et al. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies. USENIX Association, 2005.

[2] M. Abd-El-Malek, et al. Fault-scalable Byzantine fault-tolerant services. ACM Symposium on Operating System Principles. ACM, 2005.

[3] A. Adya, et al. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. Symposium on Operating Systems Design and Implementation. USENIX Association, 2002.

[4] M. K. Aguilera, et al. Sinfonia: a new paradigm for building scalable distributed systems. ACM Symposium on Operating System Principles. ACM, 2007.

[5] D. C. Anderson, et al. Interposed request routing for scalable network storage. Symposium on Operating Systems Design and Implementation, 2000.

[6] T. E. Anderson, et al. Serverless network file systems. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **29**(5):109–126, 1995.

[7] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.

[8] B. Callaghan, et al. *RFC 1813 - NFS version 3 protocol specification*, RFC–1813. Network Working Group, June 1995.

[9] A. L. Chervenak, et al. Protecting file systems: a survey of backup techniques. Joint NASA and IEEE Mass Storage Conference, 1998.

[10] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.

[11] M. Eisler, et al. Data ONTAP GX: a scalable storage cluster. Conference on File and Storage Technologies, 2007.

[12] D. Ellard, et al. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies. USENIX Association, 2003.

[13] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003.

[14] G. A. Gibson, et al. A cost-effective, high-bandwidth storage architecture. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.

[15] J. N. Gray. Notes on data base operating systems. In , volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[16] J. Hendricks, et al. *Improving small file performance in object-based storage*. Technical report CMU-PDL-06-104. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, May 2006.

[17] D. Hitz, et al. File system design for an NFS file server appliance. Winter USENIX Technical Conference. USENIX Association, 1994.

[18] J. H. Howard, et al. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, **6**(1):51–81. ACM, February 1988.

[19] W. Katsurashima, et al. NAS switch: a novel CIFS server virtualization. IEEE Symposium on Mass Storage Systems. IEEE, 7–10 April 2003.

[20] A. J. Klosterman and G. R. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU–CS–02–183. Carnegie Mellon University, October 2002.

[21] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, **16**(2):133–169. ACM Press, May 1998.

[22] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.

[23] A. W. Leung, et al. Measurement and analysis of large-scale network file system workloads. USENIX Annual Technical Conference. USENIX Association, 2008.

[24] Lustre, Apr 2006. http://www.lustre.org/.

[25] M. K. McKusick. Running 'fsck' in the background. BSDCon Conference, 2002.

[26] M. N. Nelson, et al. Caching in the sprite network file system. *Transactions on Computer Systems*, **6**(1):134–154. ACM, February 1988.

[27] J. Norton, et al. *Common Internet File System (CIFS) Technical Reference*. SNIA, 12–12 March 2002.

[28] When to Use Transactional NTFS, Apr 2006. http://msdn.microsoft.com/library/en−us/fileio/fs/when_to_use_transactional_ntfs.asp.

[29] S. V. Patil, et al. GIGA+: Scalable Directories for Shared File Systems. ACM Symposium on Principles of Distributed Computing. ACM, 2007.

[30] H. Patterson, et al. SnapMirror: file system based asynchronous mirroring for disaster recovery. Conference on File and Storage Technologies. USENIX Association, 2002.

[31] Reiser4 Transaction Design Document, Apr 2006. http://www.namesys.com/txn-doc.html/.

[32] F. Schmuck and R. Haskin. GPFS: a shared-disk file system for large computing clusters. Conference on File and Storage Technologies. USENIX Association, 2002.

[33] S. Shepler, et al. *Network file system (NFS) version 4 protocol*, RFC–3530. Network Working Group, April 2003.

[34] S. Sinnamohideen, et al. *A Transparently-Scalable Metadata Service for the Ursa Minor Storage System*. Technical report CMU-PDL-10-102. Parallel Data Laboratory, Carnegie Mellon University, March 2010.

[35] SPEC SFS97 R1 V3.0 Documentation, Jan 2010. http://www.spec.org/sfs97r1/.

[36] C. A. Thekkath, et al. Frangipani: a scalable distributed file system. ACM Symposium on Operating System Principles. Published as *Operating Systems Review*, **31**(5):224–237. ACM, 1997.

[37] S. A. Weil, et al. Ceph: A scalable, high-performance distributed file system. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006.

[38] B. Welch, et al. Scalable performance of the Panasas file system. Conference on File and Storage Technologies. USENIX Association, 2008.

[39] K. G. Yocum, et al. Anypoint: extensible transport switching on the edge. USENIX Symposium on Internet Technologies and Systems. USENIX Association, 2003.

FlashVM: Virtual Memory Management on Flash

Mohit Saxena and Michael M. Swift
University of Wisconsin-Madison
{msaxena,swift}@cs.wisc.edu

Abstract

With the decreasing price of flash memory, systems will increasingly use solid-state storage for virtual-memory paging rather than disks. *FlashVM* is a system architecture and a core virtual memory subsystem built in the Linux kernel that uses dedicated flash for paging.

FlashVM focuses on three major design goals for memory management on flash: high performance, reduced flash wear out for improved reliability, and efficient garbage collection. FlashVM modifies the paging system along code paths for allocating, reading and writing back pages to optimize for the performance characteristics of flash. It also reduces the number of page writes using zero-page sharing and page sampling that prioritize the eviction of clean pages. In addition, we present the first comprehensive description of the usage of the *discard* command on a real flash device and show two enhancements to provide fast online garbage collection of free VM pages.

Overall, the FlashVM system provides up to 94% reduction in application execution time and is four times more responsive than swapping to disk. Furthermore, it improves reliability by writing up to 93% fewer pages than Linux, and provides a garbage collection mechanism that is up to 10 times faster than Linux with discard support.

1 Introduction

Flash memory is one of the largest changes to storage in recent history. Solid-state disks (SSDs), composed of multiple flash chips, provide the abstraction of a block device to the operating system similar to magnetic disks. This abstraction favors the use of flash as a replacement for disk storage due to its faster access speeds and lower energy consumption [1, 25, 33].

In this paper, we present FlashVM, a system architecture and a core virtual memory subsystem built in the Linux kernel for managing flash-backed virtual memory. FlashVM extends a traditional system organization with dedicated flash for swapping virtual memory pages. Dedicated flash allows FlashVM software to use semantic information, such as the knowledge about free blocks, that is not available within an SSD. Furthermore, dedicating flash to virtual memory is economically attractive, because small quantities can be purchased for a few dollars. In contrast, disks ship only in large sizes at higher initial costs.

The design of FlashVM focuses on three aspects of flash: performance, reliability, and garbage collection. We analyze the existing Linux virtual memory implementation and modify it to account for flash characteristics. FlashVM modifies the paging system on code paths affected by the performance differences between flash and disk: on the read path during a page fault, and on the write path when pages are evicted from memory. On the read path, FlashVM leverages the low seek time on flash to prefetch more useful pages. The Linux VM prefetches eight physically contiguous pages to minimize disk seeks. FlashVM uses stride prefetching to minimize memory pollution with unwanted pages at a negligible cost of seeking on flash. This results in a reduction in the number of page faults and improves the application execution time. On the write path, FlashVM throttles the page write-back rate at a finer granularity than Linux. This allows better congestion control of paging traffic to flash and improved page fault latencies for various application workloads.

The write path also affects the *reliability* of FlashVM. Flash memory suffers from wear out, in that a single block of storage can only be written a finite number of times. FlashVM uses *zero-page sharing* to avoid writing empty pages and uses *page sampling*, which probabilistically skips over dirty pages to prioritize the replacement of clean pages. Both techniques reduce the number of page writes to the flash device, resulting in improved reliability for FlashVM.

The third focus of FlashVM is efficient *garbage collection*, which affects both reliability and performance. Modern SSDs provide a *discard* command (also called *trim*) for the OS to notify the device when blocks no longer contain valid data [30]. We present the first comprehensive description of the semantics, usage and performance characteristics of the discard command on a real SSD. In addition, we propose two different techniques, merged and dummy discards, to optimize the use of the discard command for online garbage collection of free VM page clusters on the swap device. Merged discard batches requests for discarding multiple page clusters in a single discard command. Alternatively, dummy discards implicitly notify the device about free VM pages by overwriting a logical flash block.

We evaluate the costs and benefits of each of these design techniques for FlashVM for memory-intensive applications representing a variety of computing environments including netbooks, desktops and distributed clusters. We show that FlashVM can benefit a variety of workloads including image manipulation, model checking, transaction processing, and large key-value stores. Our results show that:

- FlashVM provides up to 94% reduction in application execution time and up to 84% savings in memory required to achieve the same performance as swapping to disk. FlashVM also scales with increased degree of multiprogramming and provides up to four times faster response time to revive suspended applications.
- FlashVM provides better flash reliability than Linux by reducing the number of page writes to the swap device. It uses zero-page sharing and dirty page sampling for preferential eviction of clean pages, which result in up to 93% and 14% fewer page writes respectively.
- FlashVM optimizes the performance for garbage collection of free VM pages using merged and dummy discard operations, which are up to 10 times faster than Linux with discard support and only 15% slower than Linux without discard support.

The remainder of the paper is structured as follows. Section 2 describes the target environments and makes a case for FlashVM. Section 3 presents FlashVM design overview and challenges. We describe the design in Sections 4.1, covering performance; 4.2 covering reliability; and 4.3 on efficient garbage collection using the discard command. We evaluate the FlashVM design techniques in Section 5, and finish with related work and conclusions.

2 Motivation

Application working-set sizes have grown many-fold in the last decade, driving the demand for cost-effective mechanisms to improve memory performance. In this section, we motivate the use of flash-backed virtual memory by comparing it to DRAM and disk, and noting the workload environments that benefit the most.

2.1 Why FlashVM?

Fast and cheap flash memory has become ubiquitous. More than 2 *exabytes* of flash memory were manufactured worldwide in 2008. Table 1 compares the price and performance characteristics of NAND flash memory with DRAM and disk. Flash price and performance are between DRAM and disk, and about five times cheaper

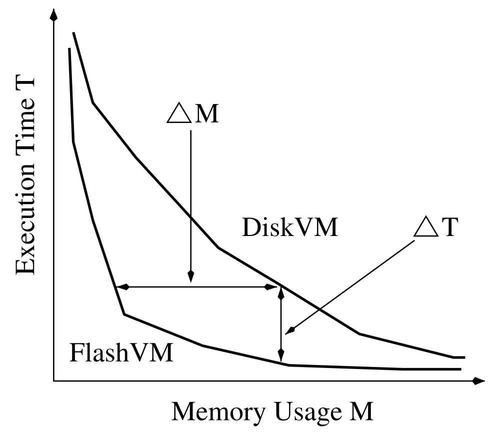


Figure 1: **Cost/Benefit Analysis:** *Application execution time plot comparing the performance of disk and flash backed virtual memory with variable main memory sizes. ΔM is the memory savings to achieve the same performance as disk and ΔT is the performance improvement with FlashVM for the same memory size.*

Device	Read Latencies (μs)		Write Latencies (μs)		Price
	Random	Seq	Random	Seq	$/GB
DRAM	0.05	0.05	0.05	0.05	$15
Flash	100	85	2,000	200-500	$3
Disk	5,000	500	5,000	500	$0.3

Table 1: **Device Attributes:** *Comparing DRAM, NAND flash memory and magnetic disk. Both price and performance for flash lie between DRAM and disk (some values are roughly defined for comparison purposes only).*

than DRAM and an order of magnitude faster than disk. Furthermore, flash power consumption (0.06 W when idle and 0.15–2 W when active) is significantly lower than both DRAM (4–5 W/DIMM) and disk (13–18 W). These features of flash motivate its adoption as second-level memory between DRAM and disk.

Figure 1 illustrates a cost/benefit analysis in the form of two simplified curves (not to scale) showing the execution times for an application in two different systems configured with variable memory sizes, and either disk or flash for swapping. This figure shows two benefits to applications when they must page. First, FlashVM results in faster execution for approximately the same system price without provisioning additional DRAM, as flash is five times cheaper than DRAM. This performance gain is shown in Figure 1 as ΔT along the vertical axis. Second, a FlashVM system can achieve performance similar to swapping to disk with lower main memory requirements. This occurs because page faults are an order of magnitude faster with flash than disk: a program can achieve the same performance with less memory by faulting more frequently to FlashVM. This reduction in memory-resident working set is shown as ΔM along the horizontal axis.

However, the adoption of flash is fundamentally an economic decision, as performance can also be improved by purchasing additional DRAM or a faster disk. Thus, careful analysis is required to configure the balance of DRAM and flash memory capacities that is optimal for the target environment in terms of both price and performance.

2.2 Where FlashVM?

Both the price/performance gains for FlashVM are strongly dependent on the workload characteristics and the target environment. In this paper, we target FlashVM against the following workloads and environments:

Netbook/Desktop. Netbooks and desktops are usually constrained with cost, the number of DIMM slots for DRAM modules and DRAM power-consumption. In these environments, the large capacity of disks is still desirable. Memory-intensive workloads, such as image manipulation, video encoding, or even opening multiple tabs in a single web browser instance can consume hundreds of megabytes or gigabytes of memory in a few minutes of usage [4], thereby causing the system to page. Furthermore, end users often run multiple programs, leading to competition for memory. FlashVM meets the requirements of such workloads and environments with faster performance that scales with increased multiprogramming.

Distributed Clusters. Data-intensive workloads such as virtualized services, key-value stores and web caches have often resorted to virtual or distributed memory solutions. For example, the popular *memcached* [17] is used to increase the aggregate memory bandwidth. While disk access is too slow to support page faults during request processing, flash access times allow a moderate number of accesses. Fast swapping can also benefit virtual machine deployments, which are often constrained by the main memory capacities available on commonly deployed cheap servers [19]. Virtual machine monitors can host more virtual machines with support for swapping out nearly the entire guest physical memory. In such cluster scenarios, hybrid alternatives similar to FlashVM that incorporate DRAM and large amounts of flash are an attractive means to provide large memory capacities cheaply [6].

3 Design Overview

The FlashVM design, shown in Figure 2, consists of dedicated flash for swapping out virtual memory pages and changes to the Linux virtual memory hierarchy that optimize for the characteristics of flash. We target FlashVM against NAND flash, which has lower prices and better write performance than the alternative, NOR flash. We propose that future systems be built with a small multiple of DRAM size as flash that is attached to the motherboard for the express purpose of supporting virtual memory.

Flash Management. Existing solid-state disks (SSD) manage NAND flash memory packages internally for emulating disks [1]. Because flash devices do not support re-writing data in place, SSDs rely on a translation layer to implement block address translation, wear leveling and garbage collection of free blocks. The translation from this layer raises three problems not present with disks: write amplification, low reliability, and aging.

Write amplification occurs when writing a single block causes the SSD to re-write multiple blocks, and leads to expensive read-modify-erase-write cycles (erase latency for a typical 128–512 KB flash block is as high as 2 milliseconds) [1, 26]. SSDs may exhibit *low reliability* because a single block may only be re-written a finite number of times. This limit is around 10,000 and is decreasing with the increase in capacity and density of MLC flash devices. Above this limit, devices may exhibit unacceptably high bit error rates [18]. For a 16 GB SSD written at its full bandwidth of 200 MB/sec, errors may arise in as little as a few weeks. Furthermore, SSDs exhibit *aging* after extensive use because fewer clean blocks are available for writing [24, 26]. This can lead to performance degradation, as the device continuously copies data to clean pages. The FlashVM design leverages semantic information only available within the operating system, such as locality of memory references, page similarity and knowledge about deleted blocks, to address these three problems.

FlashVM Architecture. The FlashVM architecture targets dedicated flash for virtual memory paging. Dedicating flash for virtual memory has two distinct advantages over traditional disk-based swapping. First, dedicated flash is cheaper than traditional disk-based swap devices in price per byte only for small capacities required for virtual memory. A 4 GB MLC NAND flash chip costs less than $6, while the cheapest IDE/SCSI disk of similar size costs no less than $24 [5, 32]. Similarly, the more common SATA/SAS disks do not scale down to capacities smaller than 36 GB, and even then are far more expensive than flash. Furthermore, the premium for managed flash, which includes a translation layer, as compared to raw flash chips is dropping rapidly as SSDs mature. Second, dedicating flash for virtual memory minimizes the interference between the file system I/O and virtual-memory paging traffic. We prototype FlashVM using MLC NAND flash-based solid-state disks connected over a SATA interface.

FlashVM Software. The FlashVM memory manager, shown in Figure 2, is an enhanced version of the memory management subsystem in the Linux 2.6.28 kernel.

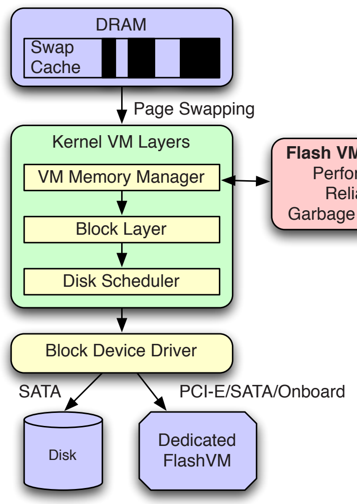


Figure 2: **FlashVM Memory Hierarchy:** *FlashVM manager controls the allocation, read and write-back of pages swapped out from main memory. It hands the pages to the block layer for conversion into block I/O requests, which are submitted to the dedicated flash device.*

Since NAND flash is internally organized as a block device, the FlashVM manager enqueues the evicted pages at the block layer for scheduling. The block layer is responsible for the conversion of pages into block I/O requests submitted to the device driver. At a high-level, FlashVM manages the non-ideal characteristics of flash and exploits its useful attributes. In particular, our design goals are:

- *High performance* by leveraging the unique performance characteristics of flash, such as fast random reads (discussed in Section 4.1).
- *Improved reliability* by reducing the number of page writes to the swap device (discussed in Section 4.2).
- *Efficient garbage collection* of free VM pages by delaying, merging, and virtualizing discard operations (discussed in Section 4.3).

The FlashVM implementation is not a singular addition to the Linux VM. As flash touches on many aspects of performance, FlashVM modifies most components of the Linux virtual memory hierarchy, including the swap-management subsystem, the memory allocator, the page scanner, the page-replacement and prefetching algorithms, the block layer and the SCSI subsystem. In the next section, we identify and describe our changes to each of these subsystems for achieving the different FlashVM design goals.

4 Design and Implementation

This section discusses the FlashVM implementation to improve performance, reliability, and to provide efficient garbage collection.

4.1 FlashVM Performance

Challenges. The virtual-memory systems of most operating systems were developed with the assumption that disks are the only swap device. While disks exhibit a range of performance, their fundamental characteristic is the speed difference between random and sequential access. In contrast, flash devices have a different set of performance characteristics, such as fast random reads, high sequential bandwidths, low access and seek costs, and slower writes than reads. For each code path in the VM hierarchy affected by these differences between flash and disk, we describe our analysis for tuning parameters and our implementation for optimizing performance with flash. We analyze three VM mechanisms: *page pre-cleaning*, *page clustering* and *disk scheduling*, and re-implement *page scanning* and *prefetching* algorithms.

4.1.1 Page Write-Back

Swapping to flash changes the performance of writing back dirty pages. Similar to disk, random writes to flash are costlier than sequential writes. However, random reads are inexpensive, so write-back should optimize for write locality rather than read locality. FlashVM accomplishes this by leveraging the *page pre-cleaning* and *clustering* mechanisms in Linux to reduce page write overheads.

Pre-cleaning. Page pre-cleaning is the act of eagerly swapping out dirty pages before new pages are needed. The Linux page-out daemon *kswapd* runs periodically to write out 32 pages from the list of inactive pages. The higher write bandwidth of flash allows FlashVM write pages more aggressively, and without competing file system traffic, use more I/O bandwidth.

Thus, we investigate writing more pages at a time to achieve sequential write performance on flash. For disks, pre-cleaning more pages interferes with high-priority reads to service a fault. However, with flash, the lower access latency and higher bandwidths enable more aggressive pre-cleaning without affecting the latency for handling a page-fault.

Clustering. The Linux clustering mechanism assigns locations in the swap device to pages as they are written out. To avoid random writes, Linux allocates *clusters*, which are contiguous ranges of *page slots*. When a cluster has been filled, Linux scans for a free cluster from the start of the swap space. This reduces seek overheads on disk by consolidating paging traffic near the beginning of the swap space.

We analyze FlashVM performance for a variety of cluster sizes from 8 KB to 4096 KB. In addition, for clusters at least the size of an erase block, we align clusters

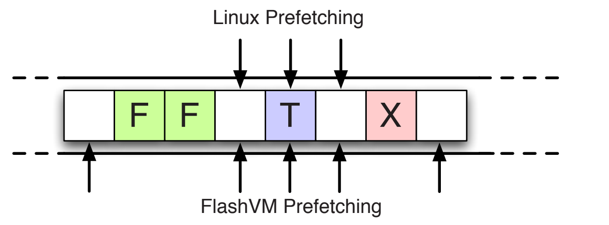


Figure 3: **Virtual Memory Prefetching**: *Linux reads-around an aligned block of pages consisting of the target page and delimited by free or bad blocks to minimize disk seeks. FlashVM skips the free and bad blocks by seeking to the next allocated page and reads all valid pages. (T, F and X represent target, free and bad blocks on disk respectively; unfilled boxes represent the allocated pages).*

with erase-block boundaries to ensure minimum amount of data must be erased.

4.1.2 Page Scanning

A virtual memory system must ensure that the rate at which it selects pages for eviction matches the write bandwidth of the swap device. Pages are selected by two code paths: memory reclaim during page allocation; and the page-out daemon that scan the inactive page list for victim pages. The Linux VM subsystem balances the rate of scanning with the rate of write-back to match the bandwidth of the swap device. If the scanning rate is too high, Linux throttles page write-backs by waiting for up to 20–100 milliseconds or until a write completes. This timeout, appropriate for disk, is more than two orders of magnitude greater than flash access latencies.

FlashVM controls write throttling at a much finer granularity of a system *jiffy* (one clock tick). Since multiple page writes in a full erase block on flash take up to two milliseconds, FlashVM times-out for about one millisecond on our system. These timeouts do not execute frequently, but have a large impact on the average page fault latency [29]. This enables FlashVM to maintain higher utilization of paging bandwidth and speeds up the code path for write-back when reclaiming memory.

4.1.3 Prefetching on Page Fault

Operating systems prefetch pages after a page fault to benefit from the sequential read bandwidth of the device [12]. The existing Linux prefetch mechanism reads in up to 8 pages contiguous on disk around the target page. Prefetching is limited by the presence of free or bad page slots that represent bad blocks on disk. As shown in Figure 3, these page slots delimit the start or the end of the prefetched pages. On disk, this approach avoids the extra cost of seeking around free and bad pages, but often leads to fetching fewer than 8 pages.

FlashVM leverages fast random reads on flash with two different prefetching mechanisms. First, FlashVM seeks over the free/bad pages when prefetching to retrieve a full set of valid pages. Thus, the fast random access of flash medium enables FlashVM to bring in more pages with spatial locality than native Linux.

Fast random access on flash also allows prefetching of more distant pages with temporal locality, such as stride prefetching. FlashVM records the offsets between the current target page address and the last two faulting addresses. Using these two offsets, FlashVM computes the strides for the next two pages expected to be referenced in the future. Compared to prefetching adjacent pages, stride prefetching reduces memory pollution by reading the pages that are more likely to be referenced soon.

We implement stride prefetching to work in conjunction with contiguous prefetching: FlashVM first reads pages contiguous to the target page and then prefetches stride pages. We find that fetching too many stride pages increases the average page fault latency, so we limit the stride to two pages. These two prefetching schemes result in a reduction in the number of page faults and improve the total execution time for paging.

4.1.4 Disk Scheduling

The Linux VM subsystem submits page read and write requests to the block-layer I/O scheduler. The choice of the I/O scheduler affects scalability with multiprogrammed workloads, as the scheduler selects the order in which requests from different processes are sent to the swap device.

Existing Linux I/O schedulers optimize performance by (i) merging adjacent requests, (ii) reordering requests to minimize seeks and to prioritize requests, and (iii) delaying requests to allow a process to submit new requests. Work-conserving schedulers, such as the NOOP and deadline schedulers in Linux, submit pending requests to the device as soon as the prior request completes. In contrast, non-work-conserving schedulers may delay requests for up to 2–3 ms to wait for new requests with better locality or to distribute I/O bandwidth fairly between processes [9]. However, these schedulers optimize for the performance characteristics of disks, where seek is the dominant cost of I/O. We therefore analyze the impact of different I/O schedulers on FlashVM.

The Linux VM system tends to batch multiple read requests on a page fault for prefetching, and multiple write requests for clustering evicted pages. Thus, paging traffic is more regular than file system workloads in general. Further, delaying requests for locality can lead to lower device utilization on flash, where random access is only a small component of the page transfer cost. Thus, we analyze the performance impact of work conservation when scheduling paging traffic for FlashVM.

4.2 FlashVM Reliability

Challenges. As flash geometry shrinks and MLC technology packs more bits into each memory cell, the prices of flash devices have dropped significantly. Unfortunately, so has the *erasure limit* per flash block. A single flash block can typically undergo between 10,000 and 100,000 erase cycles, before it can no longer reliably store data. Modern SSDs and flash devices use internal wear-leveling to spread writes across all flash blocks. However, the bit error rates of these devices can still become unacceptably high once the erasure limit is reached [18]. As the virtual memory paging traffic may stress flash storage, FlashVM specially manages page writes to improve device reliability. It exploits the information available in the OS about the state and content of a page by employing *page sampling* and *page sharing* respectively. FlashVM aims to reduce the number of page writes and prolong the lifetime of the flash device dedicated for swapping.

4.2.1 Page Sampling

Linux reclaims free memory by evicting inactive pages in a least-recently-used order. Clean pages are simply added to the free list, while reclaiming dirty pages requires writing them back to the swap device.

FlashVM modifies the Linux page replacement algorithm by prioritizing the reclaim of younger *clean* pages over older *dirty* pages. While scanning for pages to reclaim, FlashVM skips dirty pages with a probability dependent on the rate of pre-cleaning. This policy increases the number of clean pages that are reclaimed during each scan, and thus reduces the overall number of writes to the flash device.

The optimal rate for sampling dirty pages is strongly related to the memory reference pattern of the application. For applications with read-mostly page references, FlashVM can find more clean pages to reclaim. However, for applications that frequently modify many pages, skipping dirty pages for write-back leads to more frequent page faults, because younger clean pages must be evicted.

FlashVM addresses workload variations with adaptive page sampling: the probability of skipping a dirty page also depends on the write rate of the application. FlashVM predicts the average write rate by maintaining a moving average of the time interval t_n for writing *n* dirty pages. When the application writes to few pages and t_n is large, FlashVM more aggressively skips dirty pages. For applications that frequently modify many pages, FlashVM reduces the page sampling probability unless *n* pages have been swapped out. The balance between the rate of page sampling and page writes is adapted to provide a smooth tradeoff between device lifetime and application performance.

4.2.2 Page Sharing

The Linux VM system writes back pages evicted from the LRU inactive list without any knowledge of page content. This may result in writing many pages to the flash device that share the same content. Detecting identical or similar pages may require heavyweight techniques like explicitly tracking changes to each and every page by using transparent page sharing [3] or content-based page sharing by maintaining hash signatures for all pages [8]. These techniques reduce the memory-resident footprint and are orthogonal to the problem of reducing the number of page write-backs.

We implement a limited form of content-based sharing in FlashVM by detecting the swap-out of *zero pages* (pages that contain only zero bytes). Zero pages form a significant fraction of the memory-footprint of some application workloads [8]. FlashVM intercepts paging requests for all zero pages. A swap-out request sets a zero flag in the corresponding page slot in the swap map, and skips submitting a block I/O request. Similarly, a swap-in request verifies the zero flag, which if found set, allocates a zero page in the address space of the application. This extremely lightweight page sharing mechanism saves both the memory allocated for zero pages in the main-memory swap cache and the number of page write-backs to the flash device.

4.3 FlashVM Garbage Collection

Challenges. Flash devices cannot overwrite data in place. Instead, they must first erase a large flash block (128–512 KB), a slow operation, and then write to pages within the erased block. Lack of sufficient pre-erased blocks may result in copying multiple flash blocks for a single page write to: (i) replenish the pool of clean blocks, and (ii) ensure uniform wear across all blocks. Therefore, flash performance and overhead of wear management are strongly dependent on the number of clean blocks available within the flash device. For example, high-end enterprise SSDs can suffer up to 85% drop in write performance after extensive use [24, 26]. As a result, efficient garbage collection of clean blocks is necessary for flash devices, analogous to the problem of segment cleaning for log-structured file systems [28].

For virtual memory, sustained paging can quickly *age* the dedicated flash device by filling up all free blocks. When FlashVM overwrites a block, the device can reclaim the storage previously occupied by that block. However, only the VM system has knowledge about empty (free) page clusters. These clusters consist of page slots in the swap map belonging to terminated processes, dirty pages that have been read into the memory and all blocks on the swap device after a reboot. Thus, a flash device that implements internal garbage collection or wear-leveling may unnecessarily copy stale data, reducing performance and reliability when not informed about invalid pages.

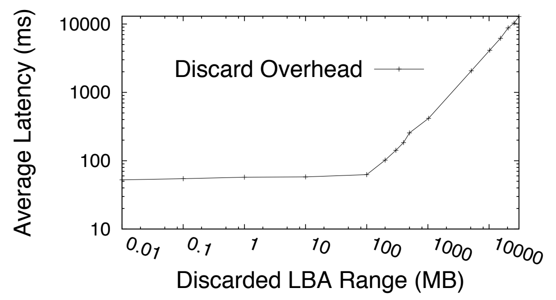


Figure 4: **Discard Overhead**: *Impact of the number of blocks discarded on the average latency of a single discard command. Both x-axis and y-axis are log-scale.*

FlashVM addresses this problem by explicitly notifying the flash device of such pages by using the *discard* command (also called *trim* introduced in the recent SATA SSDs [30]). The discard command has the following semantics:

discard(*dev*, *rangelist*[(sector, nsectors), ...])

where *rangelist* is the list of logical block address ranges to be discarded on the flash device *dev*. Each block range is represented as a pair of the starting sector address and the number of following sectors.

Free blocks can be discarded offline by first flushing all in-flight read/write requests to the flash device, and then wiping the requested logical address space. However, offline discard typically offers very coarse-grain functionality, for example in the form of periodic disk scrubbing or disk format operations [20]. Therefore, FlashVM employs online cleaning that discards a smaller range of free flash page clusters at runtime.

Linux implements rudimentary support for online cleaning in recent kernel versions starting in 2.6.28. When it finds 1 MB of free contiguous pages, it submits a single discard request for the corresponding page cluster. However, Linux does not fully support discard yet: the block layer breaks discard requests into smaller requests of no more than 256 sectors, while the ATA disk driver ignores them. Thus, the existing Linux VM is not able to actually discard the free page clusters. FlashVM instead bypasses the block and ATA driver layers and sends discard commands directly to the flash device through the SCSI layer [14, 16]. Thus, FlashVM has the ability to discard any number of sectors in a single command.

We next present an experimental analysis of the cost of discard on current flash devices. Based on these results, which show that discard is expensive, we describe two different techniques that FlashVM uses to improve the performance of garbage collection: *merged discard* and *dummy discard*.

4.3.1 Discard Cost

We measure the latency of discard operations on the OCZ-Vertex SSD, which uses the Indilinx flash controller used by many SSD manufacturers. Figure 4 shows the overheads for discard commands issued over block address ranges of different sizes. Based on Figure 4, we infer the cost of a single discard command for cleaning *B* flash blocks in one or more address ranges, each having an average utilization *u* of valid (not previously cleaned) pages:

$$cost_M = \begin{cases} c_o & \text{if } B \leq B_o \\ c_o + m \cdot u \cdot (B - B_o) & \text{otherwise} \end{cases}$$

In this equation, c_o is the fixed cost of discarding up to B_o blocks and *m* is the marginal cost of discarding each additional block. *Interestingly, the fixed cost of a single discard command is 55 milliseconds!* We speculate that this overhead occurs because the SSD controller performs multiple block erase operations on different flash channels when actually servicing a discard command [15]. The use of an on-board RAM buffer conceals the linear increase only up to a range of B_o blocks lying between 10–100 megabytes.

The cost of discard is exacerbated by the effect of command queuing: the ATA specification defines the discard commands as *untagged*, requiring that every discard be followed by an I/O barrier that stalls the request queue while it is being serviced. Thus, the long latency of discards requires that FlashVM optimize the use of the command, as the overhead incurred may outweigh the performance and reliability benefits of discarding free blocks.

4.3.2 Merged Discard

The first optimization technique that FlashVM uses is *merged discard*. Linux limits the size of each discard command sent to the device to 128 KB. However, as shown in Figure 4, discards get cheaper per byte as range sizes increase. Therefore, FlashVM opportunistically discards larger block address ranges. Rather than discard pages on every scan of the swap map, FlashVM defers the operation and batches discards from multiple scans. It discards the largest possible range list of free pages up to a size of 100 megabytes in a single command.

This approach has three major benefits. First, delaying discards reduces the overhead of scanning the swap map. Second, merging discard requests amortizes the fixed discard cost c_o over multiple block address ranges.

Third, FlashVM merges requests for fragmented and non-contiguous block ranges. In contrast, the I/O scheduler only merges contiguous read/write requests.

4.3.3 Dummy Discard

Discard is only useful when it creates free blocks that can later be used for writes. Overwriting a block *also* causes the SSD to discard the old block contents, but without paying the high fixed costs of the discard command. Furthermore, overwriting a free block removes some of the benefit of discarding to maintain a pool of empty blocks. Therefore, FlashVM implements *dummy discard* to avoid a discard operation when unnecessary.

Dummy discard elides a discard operation if the block is likely to be overwritten soon. This operation implicitly informs the device that the old block is no longer valid and can be asynchronously garbage collected without incurring the fixed cost of a discard command. As FlashVM only writes back page clusters that are integral multiples of erase-block units, no data from partial blocks needs to be relocated. Thus, the cost for a dummy discard is effectively zero.

Unlike merged discards, dummy discards do not make new clean blocks available. Rather, they avoid an ineffective discard, and can therefore only replace a fraction of all discard operations. FlashVM must therefore decide when to use each of the two techniques. Ideally, the number of pages FlashVM discards using each operation depends on the available number of clean blocks, the ratio of their costs and the rate of allocating free page clusters. Upcoming and high-end enterprise SSDs expose the number of clean blocks available within the device [24]. In the absence of such functionality, FlashVM predicts the rate of allocation by estimating the expected time interval t_s between two successive scans for finding a free page cluster. When the system scans frequently, recently freed blocks are overwritten soon, so FlashVM avoids the extra cost of discarding the old contents. When scans occur rarely, discarded clusters remain free for an extended period and benefit garbage collection. Thus, when t_s is small, FlashVM uses dummy discards, and otherwise applies merged discards to a free page cluster in the swap map.

4.4 Summary

FlashVM architecture improves performance, reliability and garbage collection overheads for paging to dedicated flash. Some of the techniques incorporated in FlashVM, such as zero-page sharing, also benefit disk-backed virtual memory. However, the benefit of sharing is more prominent for flash, as it provides both improved performance and reliability.

While FlashVM is designed for managed flash, much of its design is applicable to unmanaged flash as well. In such a system, FlashVM would take more control over garbage collection. With information about the state of pages, it could more effectively clean free pages without an expensive discard operation. Finally, this design avoids the cost of storing persistent mappings of logical block addresses to physical flash locations, as virtual memory is inherently volatile.

Device	Sequential (MB/s)		Random 4K-IO/s		Latency
	Read	Write	Read	Write	ms
Seagate Disk	80	68	120-300/s		4-5
IBM SSD	69	20	7K/s	66/s	0.2
OCZ SSD	230	80	5.5K/s	4.6K/s	0.2
Intel SSD	250	70	35K/s	3.3K/s	0.1

Table 2: **Device Characteristics:** *First-generation IBM SSD is comparable to disk in read bandwidth but excels for random reads. Second-generation OCZ-Vertex and Intel SSDs provide both faster read/write bandwidths and IOPS. Write performance asymmetry is more prominent in first-generation SSDs.*

5 Evaluation

The implementation of FlashVM entails two components: changes to the virtual memory implementation in Linux and dedicated flash for swapping. We implement FlashVM by modifying the memory management subsystem and the block layer in the x86-64 Linux 2.6.28 kernel. We focus our evaluation on three key questions surrounding these components:

- How much does the FlashVM architecture of *dedicated flash* for virtual memory improve performance compared to traditional disk-based swapping?
- Does FlashVM software design improve *performance*, *reliability* via write-endurance and *garbage collection* for virtual memory management on flash?
- Is FlashVM a *cost-effective* approach to improving system price/performance for different real-world application workloads?

We first describe our experimental setup and methodology and then present our evaluation to answer these three questions in Section 5.2, 5.3 and 5.4 respectively. We answer the first question by investigating the benefits of dedicating flash for paging in Section 5.2. In Section 5.3 and 5.4, we isolate the impact of FlashVM software design by comparing against the native Linux VM implementation.

5.1 Methodology

System and Devices. We run all tests on a 2.5 GHz Intel Core 2 Quad system configured with 4 GB DDR2 DRAM and 3 MB L2 cache per core, although we reduce the amount of memory available to the OS for our tests, as and when mentioned. We compare four storage devices: an IBM first generation SSD, a trim-capable OCZ-Vertex SSD, an Intel X-25M second generation SSD, and a Seagate Barracuda 7200 RPM disk, all using native command queuing. Device characteristics are shown in Table 2.

Application Workloads. We evaluate FlashVM performance with four memory-intensive application workloads with varying working set sizes:

1. *ImageMagick* 6.3.7, resizing a large JPEG image by 500%,
2. *Spin* 5.2 [31], an LTL model checker for testing mutual exclusion and race conditions with a depth of 10 million states,
3. *pseudo-SpecJBB*, a modified SpecJBB 2005 benchmark to measure execution time for 16 concurrent data warehouses with 1 GB JVM heap size using Sun JDK 1.6.0,
4. *memcached* 1.4.1 [17], a high-performance object caching server bulk-storing or looking-up 1 million random 1 KB key-value pairs.

All workloads have a virtual memory footprint large enough to trigger paging and reach steady state for our analysis. For all our experiments, we report results averaged over five different runs. While we tested with all SSDs, we mostly present results for the second generation OCZ-Vertex and Intel SSDs for brevity.

5.2 Dedicated Flash

We evaluate the benefit of dedicating flash to virtual memory by: (i) measuring the costs of sharing storage with the file system, which arise from scheduling competing I/O traffic, and (ii) comparing the scalability of virtual memory with traditional disk-based swapping.

5.2.1 Read/Write Interference

With disk, the major cost of interference is the seeks between competing workloads. With an SSD, however, seek cost is low and the cost of interference arises from interleaving reads and writes from the file and VM systems. Although this cost occurs with disks as well, it is dominated by the overhead of seeking. We first evaluate the performance loss from interleaving, and then measure the actual amount of interleaving with FlashVM.

We use a synthetic benchmark that reads or writes a sequence of five contiguous blocks. Figure 5(a) shows I/O performance as we interleave reads and writes for disk, IBM SSD and Intel SSD. For disk, I/O performance drops from its sequential read bandwidth of 80 MB/s to 8 MB/s when the fraction of interleaved writes reaches 60% because the drive head has to be repositioned between read and write requests. On flash, I/O performance also degrades as the fraction of writes increase: IBM and Intel SSDs performance drops by 10x and 7x respectively when 60 percent of requests are writes. Thus, interleaving can severely reduce system performance.

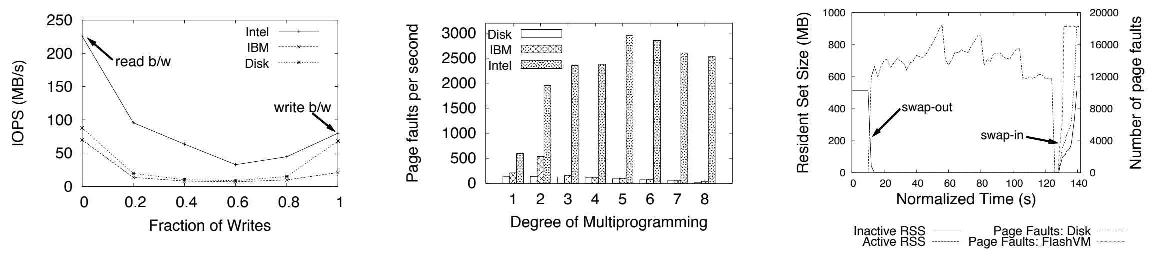
These results demonstrate the potential improvement from dedicated flash, because, unlike the file system, the VM system avoids interleaved read and write requests. To measure this ability, we traced the block I/O requests enqueued at the block layer by the VM subsystem using Linux *blktrace*. Page read and write requests are governed by prefetching and page-out operations, which batch up multiple read/write requests together. On analyzing the average length of read request streams interleaved with write requests for ImageMagick and Spin, we found that FlashVM submits long strings of read and write requests. The average length of read streams ranges between 138–169 I/O requests, and write streams are between 170–230 requests. Thus, the FlashVM system architecture benefits from dedicating flash without interleaved reads and writes from the file system.

5.2.2 Scaling Virtual Memory

Unlike flash, dedicating a disk for swapping does not scale with multiple applications contending for memory. This scalability manifests in two scenarios: increased throughput as the number of threads or programs increases, and decreased interference between programs competing for memory.

Multiprogramming. On a dedicated disk, competing programs degenerate into random page-fault I/O and high seek overheads. Figure 5(b) compares the paging throughput on different devices as we run multiple instances of ImageMagick. Performance, measured by the rate of page faults served per second, degrades for both disk and the IBM SSD with as few as 3 program instances, leading to a CPU utilization of 2–3%. For the IBM SSD, performance falls largely due to an increase in the random write traffic, which severely degrades its performance.

In contrast, we find improvement in the effective utilization of paging bandwidth on the Intel SSD with an increase in concurrency. At 5 instances, paging traffic almost saturates the device bandwidth: for each page fault FlashVM prefetches an additional 7 pages, so it reads 96 MB/s to service 3,000 page faults per second. In addition, it writes back a proportional but lower number of pages. Above 5 instances of ImageMagick, the page fault service rate drops because of increased congestion for paging traffic: CPU utilization falls from 54% with 5 concurrent programs to 44% for 8 programs, and write traffic nears the bandwidth of the device. Nevertheless, these results demonstrate that performance scales significantly as multiprogramming increases on flash when



(a) Read/Write Interference (b) Paging Throughput (c) Response Time

Figure 5: **Dedicated Flash:** *Impact of dedicating flash for VM on performance for read/write interference with file system traffic, paging throughput for multiprogrammed workloads and response time for increased memory contention.*

compared to disk. We find similar increase in paging throughput on dedicated flash for multithreaded applications like memcached. FlashVM performance and device utilization increase when more threads generate more simultaneous requests. This is much the same argument that exists for hardware multithreading to increase parallelism in the memory system.

Response Time. The second scalability benefit of dedicated flash is faster response time for demand paging when multiple applications contend for memory. Figure 5(c) depicts the phenomena frequently observed on desktops when switching to inactive applications. We model this situation with two processes each having working-set sizes of 512 MB and 1.5 GB, that contend for memory on a system configured with 1 GB of DRAM. The curves show the resident set sizes (the amount of physical memory in use by each process) and the aggregate number of page faults in the system over a time interval of 140 seconds. The first process is active for the first 10 seconds and is then swapped out by the Linux VM to accommodate the other process. When 120 seconds have elapsed, the second process terminates and the first process resumes activity.

Demand paging the first process back into memory incurs over 16,000 page faults. With disk, this takes 11.5 seconds and effectively prevents all other applications from accessing the disk. In contrast, resuming the first process takes only 3.5 seconds on flash because of substantially lower flash access latency. Thus, performance degrades much more acceptably with dedicated flash than traditional disk-based swapping, leading to better scalability as the number of processes increase.

5.3 FlashVM Software Evaluation

FlashVM enhances the native Linux virtual memory system for improved performance, reliability and garbage collection. We first analyze our optimizations to the existing VM mechanisms required for improving flash performance, followed by our enhancements for wear management and garbage collection of free blocks.

5.3.1 Performance Analysis

We analyze the performance of different codes paths that impact the paging performance of FlashVM.

Page pre-cleaning. Figure 6(a) shows the performance of FlashVM for ImageMagick, Spin and memcached as we vary the number of pages selected for write-back (pre-cleaned) on each page fault. Performance is poor when only two pages are written back because the VM system frequently scans the inactive list to reclaim pages. However, we find that performance does not improve when pre-cleaning more than 32 pages, because the overhead of scanning is effectively amortized at that point.

Page clustering. Figure 6(b) shows FlashVM performance as we vary the cluster size, the number of pages allocated contiguously on the swap device, while keeping pre-cleaning constant at 32 pages. When only two pages are allocated contiguously (cluster size is two), overhead increases because the VM system wastes time finding free space. Large cluster sizes lead to more sequential I/O, as pages are allocated sequentially within a cluster. However, our results show that above 32 page clusters, performance again stabilizes. This occurs because 32 pages, or 128 KB, is the size of a flash erase block and is enough to obtain the major benefits of sequential writes on flash. We tune FlashVM with these optimal values for pre-cleaning and cluster sizes for all our further experiments.

Congestion control. We evaluate the performance of FlashVM congestion control by comparing it against native Linux on single- and multi-threaded workloads. We separately measured the performance of changing the congestion timeout for the page allocator and for both the page allocator and the *kswapd* page-out daemon for ImageMagick. With the native Linux congestion control timeout tuned to disk access latencies, the system idles even when there is no congestion.

For single-threaded programs, reducing the timeout for the page allocator from 20ms to 1ms improved performance by 6%, and changing the timeout for *kswapd*

Read-Ahead	**Native**		**Stride**	
(# of pages)	**PF**	**Time**	**PF**	**Time**
2	139K	103.2	118K / 15%	88.5 / 14%
4	84K	96.3	70K / 17%	85.7 / 11%
8	56K	91.5	44K / 21%	85.1 / 7%
16	43K	89.0	28K / 35%	83.5 / 6%

Table 3: **VM Prefetching:** *Impact of native Linux and FlashVM prefetching on the number of page faults and application execution time for ImageMagick. (PF is number of hard page faults in thousands, Time is elapsed time in seconds, and percentage reduction and speedup are shown for the number of page faults and application execution time respectively.)*

in addition leads to a 17% performance improvement. For multithreaded workloads, performance improved 4% for page allocation and 6% for both page allocation and the *kswapd*. With multiple threads, the VM system is less likely to idle inappropriately, leading to lower benefits from a reduced congestion timeout. Nevertheless, FlashVM configures lower timeouts, which better match the latency for page access on flash.

Prefetching. Along the page-fault path, FlashVM prefetches more aggressively than Linux by reading more pages around the faulting address and fetching pages at a stride offset. Table 3 shows the benefit of these two optimizations for ImageMagick. The table lists the number of page faults and performance as we vary the number of pages read-ahead for FlashVM prefetching against native Linux prefetching, both on the Intel SSD.

We find that FlashVM outperforms Linux for all values of read-ahead. The reduction in page faults improves from 15% for two pages to 35% for 16 pages, because of an increase in the difference between the number of pages read for native Linux and FlashVM. However, the speedup decreases because performance is lost to random access that results in increased latency per page fault. More sophisticated application-directed prefetching can provide additional benefits by exploiting a more accurate knowledge of the memory reference patterns and the low seek costs on flash.

Disk Scheduling. FlashVM depends on the block layer disk schedulers for merging or re-ordering I/O requests for efficient I/O to flash. Linux has four standard schedulers, which we compare in Figure 6(c). For each scheduler, we execute 4 program instances concurrently and report the completion time of the last program. We scale the working set of the program instances to ensure relevant comparison on each individual device, so the results are not comparable across devices.

On disk, the NOOP scheduler, which only merges adjacent requests before submitting them to the block device driver in FIFO order, performs worst, because it results in long seeks between requests from different processes. The deadline scheduler, which prioritizes synchronous page faults over asynchronous writes, performs best. The other two schedulers, CFQ and anticipatory, insert delays to minimize seek overheads, and have intermediate performance.

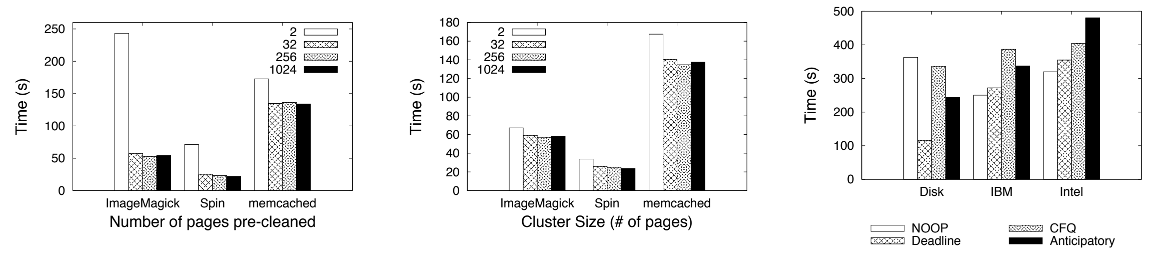
In contrast, for both flash devices the NOOP scheduler outperforms all other schedulers, outperforming CFQ and anticipatory scheduling by as much as 35% and the deadline scheduler by 10%. This occurs because there is no benefit to localizing seeks on an SSD. We find that average page access latency measured for disk increases linearly from 1 to 6 ms with increasing seek distance. In contrast, for both SSDs, seek time is constant and less than 0.2 ms even for seek distances up to several gigabytes. So, the best schedule for SSDs is to merge adjacent requests and queue up as many requests as possible to obtain the maximum bandwidth. We find that disabling delaying of requests in the anticipatory scheduler results in a 22% performance improvement, but it is still worse than NOOP. Thus, non work-conserving schedulers are not effective when swapping to flash, and scheduling as a whole is less necessary. For the remaining tests, we use the NOOP scheduler.

5.3.2 Wear Management

FlashVM reduces wear-out of flash blocks by write reduction using dirty page sampling and zero-page sharing.

Page Sampling. For ImageMagick, uniformly skipping 1 in 100 dirty pages for write back results in up to 12% reduction in writes but a 5% increase in page faults and a 7% increase in the execution time. In contrast, skipping dirty pages aggressively only when the program has a lower write rate better prioritizes the eviction of clean pages. For the same workload, adaptively skipping 1 in 20 dirty pages results in a 14% write reduction without any increase in application execution time. Thus, adaptive page sampling better reduces page writes with less affect on application performance.

Page Sharing. The number of zero pages swapped out from the inactive LRU list to the flash device is dependent on the memory-footprint of the whole system. Memcached clients bulk-store random keys, leading to few empty pages and only 1% savings in the number of page writes with zero-page sharing. In contrast, both ImageMagick and Spin result in substantial savings. ImageMagick shows up to 15% write reduction and Spin swaps up to 93% of zero pages. We find that Spin preallocates a large amount of memory and zeroes it down before the actual model verification phase begins. Zero-page sharing improves both the application execution time as well as prolongs the device lifetime by reducing the number of page writes.



(a) Page pre-cleaning (b) Page clustering (c) Disk Scheduling

Figure 6: **Performance Analysis:** *Impact of page pre-cleaning, page clustering and disk scheduling on FlashVM performance for different application workloads.*

5.3.3 Garbage Collection

FlashVM uses *merged* and *dummy* discards to optimize garbage collection of free VM pages on flash. We compare the performance of garbage collection for FlashVM against native Linux VM on an SSD. Because Linux cannot currently execute discards, we instead collect block-level I/O traces of paging traffic for different applications. The block layer breaks down the VM discard I/O requests into 128 KB discard commands, and we emulate FlashVM by merging multiple discard requests or replacing them with equivalent dummy discard operations as described in Section 4.3. Finally, we replay the processed traces on an aged trim-capable OCZ-Vertex SSD and record the total trace execution time.

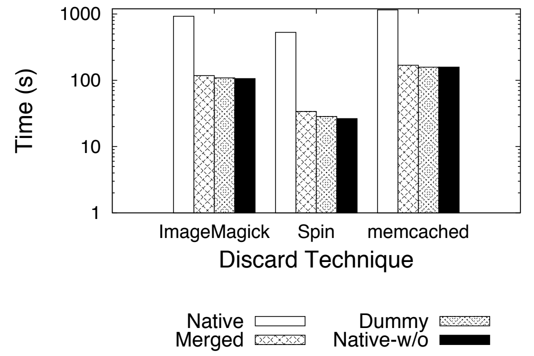


Figure 7: **Garbage Collection Performance:** *Impact of merged and dummy discards on application performance for FlashVM. y-axis is log-scale for application execution time.*

Figure 7 compares the performance of four different systems: FlashVM with merged discards over 100 MB ranges, FlashVM with dummy discards, native Linux VM with discard support and baseline Linux VM without discard support. Linux with discard is 12 times slower than the baseline system, indicating the high cost for inefficient use of the discard command. In contrast, FlashVM with merged discard, which also has the reliability benefits of Linux with discard, is only 15 *percent* slower than baseline. With the addition of adaptive dummy discards, which reduces the rate of discards when page clusters are rapidly allocated, performance is 11% slower than baseline. In all cases, the slowdown is due to the long latency of discard operations, which have little direct performance benefit. These results demonstrate that naive use of discard greatly degrades performance, while FlashVM's merged and dummy discard achieve similar reliability benefits at performance near native speeds.

Workload	**DiskVM**		**FlashVM**	
	Runtime	Mem	Const Mem Rel. Runtime	Const Runtime Rel. Memory
ImageMagick	207	814	31%	51%
Spin	209	795	11%	16%
SpecJBB	275	710	6%	19%
memcached-store	396	706	18%	60%
memcached-lookup	257	837	23%	50%

Table 4: **Cost/Benefit Analysis:** *FlashVM analysis for different memory-intensive application workloads. Systems compared are DiskVM with disk-backed VM, a FlashVM system with the same memory (Const Mem) and one with the same performance but less memory (Const Runtime). FlashVM results show the execution time and memory usage, both relative to DiskVM. Application execution time Runtime is in seconds; memory usage Mem is in megabytes.*

5.4 FlashVM Application Performance

Adoption of FlashVM is fundamentally an economic decision: a FlashVM system can perform better than a DiskVM system even when it is provisioned with more expensive DRAM. Therefore, we evaluate the performance gains and memory savings when replacing disk with flash for paging. Our results reflect estimates for absolute memory savings in megabytes.

Table 4 presents the performance and memory usage of five application workloads on three systems:

1. *DiskVM* with 1 GB memory and a dedicated disk for swapping;
2. *FlashVM - Const Mem* with the same DRAM size as DiskVM, but improved performance;
3. *FlashVM - Const Runtime* with reduced DRAM size, but same performance as DiskVM.

Our analysis in Table 4 represents configurations that correspond to the three data points shown in Figure 1. System configurations for workloads with high locality or unused memory do not page and show no benefit from FlashVM. Similarly, those with no locality or extreme memory requirements lie on the far left in Figure 1 and perform so poorly as to be unusable. Such data points are not useful for analyzing virtual memory performance. The column in Table 4 titled DiskVM shows the execution time and memory usage of the five workloads on a system swapping to disk. Under FlashVM - Const Mem and FlashVM - Const Runtime, we show the percentage reduction in the execution time and memory usage respectively, both when compared to DiskVM. The reduction in memory usage corresponds to the potential price savings by swapping to flash rather than disk for achieving similar performance.

For all applications, a FlashVM system outperforms a system configured with the same amount of DRAM and disk-backed VM (FlashVM - Const Mem against DiskVM). FlashVM's reduction in execution time varies from 69% for ImageMagick to 94% for the modified SpecJBB, a 3-16x speedup. On average, FlashVM reduces run time by 82% over DiskVM. Similarly, we find that there is a potential 60% reduction in the amount of DRAM required on the FlashVM system to achieve similar performance as DiskVM (FlashVM - Const Runtime against DiskVM). This benefit comes directly from the lower access latency and higher bandwidth of flash, and results in both price and power savings for the FlashVM system.

Overall, we find that applications with poor locality have higher memory savings because the memory saved does not substantially increase their page fault rate. In contrast, applications with good locality see proportionally more page faults from each lost memory page. Furthermore, applications also benefit differently depending on their access patterns. For example, when storing objects, *memcached* server performance improves 5x on a FlashVM system with the same memory size, but only 4.3x for a lookup workload. The memory savings differ similarly.

6 Related Work

The FlashVM design draws on past work investigating the use of solid-state memory for storage. We categorize this work into the following four classes:

Persistent Storage. Flash has most commonly been proposed as a storage system to replace disks. eNVy presented a storage system that placed flash on the memory bus with a special controller equipped with a battery-backed SRAM buffer [33]. File systems, such as YAFFS and JFFS2 [27], manage flash to hide block erase latencies and perform wear-leveling to handle bad blocks. More recently, TxFlash exposes a novel transactional interface to use flash memory by exploiting its copy-on-write nature [25]. These systems all treat flash as persistent storage, similar to a file system. In contrast, FlashVM largely ignores the non-volatile aspect of flash and instead focuses on the design of a high-performance, reliable and scalable virtual memory.

Hybrid Systems. Guided by the price and performance of flash, hybrid systems propose flash as a second-level cache between memory and disk. FlashCache uses flash as secondary file/buffer cache to provide a larger caching tier than DRAM [10]. Windows and Solaris can use USB flash drives and solid-state disks as read-optimized disk caches managed by the file system [2, 7]. All these systems treat flash as a cache of the contents on a disk and mainly exploit its performance benefits. In contrast, FlashVM treats flash as a backing store for evicted pages, accelerates both read and write operations, and provides mechanisms for improving flash reliability and efficiency of garbage collection by using the semantic information about paging only available within the OS.

Non-volatile Memory. NAND flash is the only memory technology after DRAM that has become cheap and ubiquitous in the last few decades. Other non-volatile storage class memory technologies like phase-change memory (PCM) and magneto-resistive memory (MRAM) are expected to come at par with DRAM prices by 2015 [21]. Recent proposals have advocated the use of PCM as a first-level memory placed on the memory bus alongside DRAM [11, 19]. In contrast, FlashVM adopts cheap NAND flash and incorporates it as swap space rather than memory directly addressable by user-mode programs.

Virtual Memory. Past proposals on using flash as virtual memory focused on new page-replacement schemes [23] or providing compiler-assisted, energy-efficient swap space for embedded systems [13, 22]. In contrast, FlashVM seeks more OS control for memory management on flash, while addressing three major problems for paging to dedicated flash. Further, we present the first description of the usage of the discard command on a real flash device and provide mechanisms to optimize the performance of garbage collection.

7 Conclusions

FlashVM adapts the Linux virtual memory system for the performance, reliability, and garbage collection characteristics of flash storage. In examining Linux, we find many dependencies on the performance characteristics of disks, as in the case of prefetching only adjacent pages. While the assumptions about disk performance are not made explicit, they permeate the design, particularly regarding batching of requests to reduce seek latencies and to amortize the cost of I/O. As new storage technologies with yet different performance characteristics and challenges become available, such as memristors and phase-change memory, it will be important to revisit both operating system and application designs.

Acknowledgments

Many thanks to our shepherd John Regehr and to the anonymous reviewers for their helpful comments and feedback. This work was supported by NSF Award CNS-0834473. Swift has a financial interest in Microsoft Corp.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *USENIX* (2008).

[2] ARCHER, T. MSDN Blog: Microsoft ReadyBoost. `http://blogs.msdn.com/tomarcher/archive/2006/06/02/615199.aspx`.

[3] BUGNION, E., DEVINE, S., AND ROSENBLUM, M. Disco: Running commodity operating systems on scalable multiprocessors. In *SOSP* (1997).

[4] DOTNETPERLS.COM. Memory benchmark for Web Browsers., December 2009. `http://dotnetperls.com/chrome-memory`.

[5] DRAMEXCHANGE.COM. Mlc nand flash price quote, Dec. 2009. `http://dramexchange.com/#flash`.

[6] GEAR6. Scalable hybrid memcached solutions. `http://www.gear6.com`.

[7] GREGG, B. Sun Blog: Solaris L2ARC Cache. `http://blogs.sun.com/brendan/entry/test`.

[8] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference engine: Harnessing memory redundancy in virtual machines. In *OSDI* (2008).

[9] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous IO. In *SOSP* (2001).

[10] KGIL, T., AND MUDGE, T. N. Flashcache: A nand flash memory file cache for low power web servers. In *CASES* (2006).

[11] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *ISCA* (2009).

[12] LEVY, H. M., AND LIPMAN, P. H. Virtual memory management in the VAX/VMS operating system. *Computer 15*, 3 (1982), 35–41.

[13] LI, H.-L., YANG, C.-L., AND TSENG, H.-W. Energy-aware flash memory management in virtual memory system. In *IEEE Transactions on VLSI systems* (2008).

[14] LINUX-DOCUMENTATION. SCSI Generic Driver Interface. `http://tldp.org/HOWTO/SCSI-Generic-HOWTO`.

[15] LORD, M. Author, Linux IDE Subsystem, Personal Communication. December, 2009.

[16] LORD, M. hdparm 9.27: get/set hard disk parameters. `http://linux.die.net/man/8/hdparm`.

[17] MEMCACHED. High-performance Memory Object Cache. `http://www.danga.com/memcached`.

[18] MIELKE, N., MARQUART, T., KESSENICH, J. N. W., BELGAL, H., SCHARES, E., TRIVEDI, F., GOODNESS, E., NEVILL, E., AND L.R. Bit error rate in nand flash memories. In *IEEE IRPS* (2008).

[19] MOGUL, J. C., ARGOLLO, E., SHAH, M., AND FARABOSCHI, P. Operating system support for nvm+dram hybrid main memory. In *HotOS* (2009).

[20] MSDN BLOG. Trim Support for Windows 7. `http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx`.

[21] OBJECTIVE-ANALYSIS.COM. White Paper: PCM becomes a reality., Aug. 2009. `http://www.objective-analysis.com/uploads/2009-08-03_Objective_Analysis_PCM_White_Paper.pdf`.

[22] PARK, C., LIM, J., KWON, K., LEE, J., AND MIN, S. L. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT* (2004).

[23] PARK, S., JUNG, D., KANG, J., KIM, J., AND LEE, J. CFLRU: A replacement algorithm for flash memory. In *CASES* (2006).

[24] POLTE, M., SIMSA, J., AND GIBSON, G. Enabling enterprise solid state disks performance. In *WISH* (2009).

[25] PRABHAKARAN, V., RODEHEFFER, T., AND ZHOU, L. Transactional flash. In *OSDI* (2008).

[26] RAJIMWALE, A., PRABHAKARAN, V., AND DAVIS, J. D. Block management in solid-state devices. In *USENIX* (2009).

[27] REDHAT INC. JFFS2: The Journalling Flash File System, version 2, 2003. `http://sources.redhat.com/jffs2`.

[28] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992).

[29] SAXENA, M., AND SWIFT, M. M. FlashVM: Revisiting the Virtual Memory Hierarchy. In *HotOS-XII* (2009).

[30] SHU, F., AND OBR, N. Data Set Management Commands Proposal for ATA8-ACS2, 2007. `http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc`.

[31] SPIN. LTL Model Checker, Bell Labs. `http://spinroot.com`.

[32] STREETPRICES.COM. Disk drive price quote, Dec. 2009. `http://www.streetprices.com`.

[33] WU, M., AND ZWAENEPOEL, W. eNVy: A non-volatile, main memory storage system. In *ASPLOS-VI* (1994).

Dyson: An Architecture for Extensible Wireless LANs

Rohan Murty†, Jitendra Padhye, Alec Wolman, Matt Welsh†
Microsoft Research, †Harvard University

Abstract

Dyson is a new software architecture for building customizable WLANs. While research in wireless networks has made great strides, these advancements have not seen the light of day in real WLAN deployments. One of the key reasons is that today's WLANs are not architected to embrace change. For example, system administrators cannot fine-tune the association policy for their particular environment: an administrator may know certain nodes in certain locations interfere with each other and cause a severe degradation in throughput, and hence, such associations must be avoided in the particular deployment. Dyson defines a set of APIs that allow clients and APs to send pertinent information such as radio channel conditions to a central controller. The central controller processes this information, to form a global view of the network. This global view, combined with historical information about spatial and temporal usage patterns, allows the central controller enact a rich set of policies to control the network's behavior. Dyson provides a Python-based scripting API that allows the central controller's policies to be extended for site-specific customizations and new optimizations that leverage historical knowledge. We have built a prototype implementation of Dyson, which currently runs on a 28-node testbed distributed across one floor of a typical academic building. Using this testbed, we examine various aspects of the architecture in detail, and demonstrate the ease of implementing a wide range of policies. Using Dyson, we demonstrate optimizing associations, handling VoIP clients, reserving airtime for specific users, and optimizing handoffs for mobile clients.

1 Introduction

Wireless networks are struggling to keep up with the demands of new applications, such as media streaming, voice over IP, and the increasing use of mobile devices, such as Wi-Fi enabled smartphones. Researchers have proposed numerous ways to cope with these changing demands, including new approaches to association and handoff control [22], channel allocation strategies [20, 17], and centralized packet transmission scheduling [26]. However, deploying these innovations in real wireless LANs remains a significant challenge. Enterprises wishing to roll out new applications, services, or policies in a wireless LAN are faced with ossified standards and a wide variety of software, device driver, and hardware implementations of these standards by many different vendors. Compounding this problem is the fact that existing WLAN standards generally do not allow for much customization. In this paper, we argue that it is time to rethink the architecture of wireless networks from the ground up, to enable greater observability, control, and extensibility to meet future needs.

Today, WLAN vendors offer few knobs to customize network operation. A case in point is the Microsoft campus enterprise wireless network, which uses access points supplied a single vendor. These access points are managed by a central controller, which attempts to dynamically tune the assignments for channel selection and transmit power to improve performance. Shortly after these new APs were deployed, the WLAN administrators realized that the transmit power control algorithm was not suitable for our campus. Because the algorithm was geared towards avoiding interference, many APs reduced their transmit power to such an extent that it left large holes in coverage. The controller offered no knobs to allow administrators to customize the power assignment algorithm; the only option was to disable it entirely.

In the above example, the lack of flexibility does not arise from the 802.11 standard, which is in fact quite flexible in many respects: it imposes no specific policies on association, channel assignment, or power control. The problem is that there is no agreed-upon framework to control these knobs. Moreover, there are no explicit mechanisms for stations to coordinate with each other to observe the state of the network, requiring nodes to take a purely local "every station for itself" point-of-view. This local viewpoint then encourages vendors to hard-code important algorithms into device drivers and firmware that affect WLAN performance, such as those that control AP associations, PHY data rates, and transmission power.

In this paper, we present a new WLAN architecture, called *Dyson*, that is designed to enable extensive customization and control over many aspects of network operation and performance. The Dyson architecture is designed to support *global network observation*, *deep control*, and *extensibility* to meet future needs. In Dyson, both clients and access points coordinate with the network infrastructure to provide detailed measurements on location, radio channel conditions, connectivity, and observed performance. Measurements are stored in a persistent database, allowing the infrastructure to adapt its behavior based on historical knowledge of network state. Dyson defines a set of APIs that allow clients and APs to share pertinent performance information such as packet loss rates and radio channel conditions. Dyson also defines a control interface, supported by both clients and APs, that permits the infrastructure to manage many aspects of their operation, including associations, channel selection, PHY rate, and transmission throttling. The central controller can enact a rich set of policies to control the WLAN behavior. These polices are built, customized and extended using a Python-based scripting API. Even though Dyson benefits when all clients in the WLAN support the control interface, it can still work with a mix of legacy and controllable clients.

Dyson's design hinges on the use of a centralized network controller, a common feature in recent research [22, 26, 11] and commercial [1, 2] WLANs for enterprises. None of the previous centralized WLAN systems have considered flexibility and evolvability as their primary design requirement. Dyson focuses on providing a rich set of APIs that allow the network's operation to be extended to embrace new application demands and site-specific customizations. Dyson's Python programming interface makes it easy to develop new policies and experiment with a wide range of behaviors. To demonstrate Dyson's flexibilty, we have implemented a range of policies for optimizing associations, handling VoIP clients, and reserving airtime for specific users.

This paper makes the following contributions. First, Dyson is the first wireless LAN architecture that directly addresses the need for extensibility and evolvability, leveraging both centralized control and client-side instrumentation to enable a wide range of new policies to be layered on the existing network. Second, Dyson enables more efficient use of radio spectrum by taking measurements gathered from both clients and APs into account. Third, we have implemented Dyson on a 28-node testbed distributed over one floor of an office building, which we use to evaluate the system in detail using a range of policies. We demonstrate how WLAN behavior can be easily customized via Dyson's policies to provide better performance overall.

2 Dyson Architecture

The Dyson network architecture, shown in Figure 2, consists of a number of wireless *clients*, *access points* (APs), and a single *central controller* (CC). As described below, both APs and Dyson-enabled clients report measurements to the infrastructure, which are used to construct a dynamic *network map* representing the state of the network. Measurements are also logged to a database for historical analysis, and static information on AP location and MAC addresses are stored in a separate AP database. Extensi-

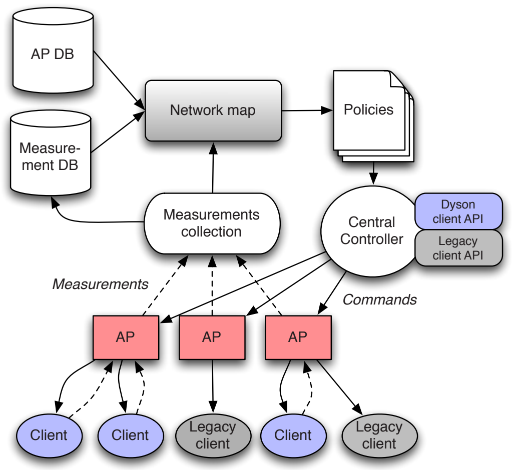


Figure 1: **The Dyson network architecture.**

bility is enabled through *policies* that are used to trigger network configuration changes via *commands* delivered by the central controller to APs and clients.

Dyson builds upon existing 802.11 standards, including CSMA MAC and the format of the data and management frames. As a result, Dyson can be implemented entirely using existing 802.11-compatible hardware. The key difference between Dyson and existing enterprise WLANs is the manner in which network management and control is performed. The Dyson architecture requires APs to be Dyson-aware. Dyson-aware clients support enhanced functionality for measurement collection and control, as described below. Legacy 802.11 clients can be supported by Dyson, although with reduced functionality. Note that Dyson-enabled clients remain compatible with legacy 802.11 networks.

The use of a central controller in enterprise WLANs is widespread.[1] For example, in Aruba [1] networks, the CC is responsible for assigning radio channels and transmission power levels to individual APs based on global observation of the network traffic. Dyson significantly augments this design by extending both observation and control to the wireless clients as well as the APs. Dyson clients are responsible for collecting periodic *measurements* of channel and traffic conditions and reporting them to the CC, as well as responding to *commands* from the CC that control many aspects of transmission parameters, as described below.

A key question that arises in this regime is how much control clients should yield to the infrastructure. At one extreme, the CC could control clients at a very fine-grained level, for example, by dictating individual packet transmission timings [26]. However, this design would require substantial control overhead, and would fail to re-

[1]Note that the CC need not be physically centralized, as this functionality can be replicated across multiple physical hosts for reliability and scalability.

spond rapidly to local changes in channel conditions (e.g., interference) at the client. In Dyson, we opt to affect control at a higher level, via channel allocations, client-AP associations and throttling. Although cruder than packet-level control, this design strikes a balance between the overhead for command issue and the ability of the network to drive towards more efficient configurations.

One implication of this design is that we assume that Dyson clients are willing participants in the system, and are capable of accurately and truthfully responding to measurement requests and commands. There is, of course, the potential that malicious or buggy clients could misbehave and degrade network performance. However, we argue that the degree of trust that Dyson places in clients is not substantially greater than that in conventional 802.11 networks, in which it must be assumed that clients correctly obey the protocol. We assume that Dyson clients are authenticated using 802.1x.

The power of the CC is derived from its global knowledge of the state of the network and ability to control both APs and clients at fine granularity. It also maintains a database to store received measurements, permitting long-term historical analysis of network performance.

A key benefit in Dyson is the ability to collect *client-side* measurements, providing the CC with greater visibility and control over the network state. Client-side information can be used to resolve sources of ambiguity that would arise with AP-only observations. Examples include detection of hidden terminals, awareness of mutual connectivity between APs and clients, and mapping channel airtime utilization. While client participation has been explored by several previous systems [11, 6], Dyson provides a flexible framework in which a wide range of policies can be specified programmatically.

2.1 Measurement collection

In Dyson, both clients and APs are responsible for collecting passive measurements on the state of the network, reporting measurements to the CC, and responding to commands issued by the CC to modify local parameters. As described above, the granularity of measurements and commands is chosen to avoid high overheads for client/CC interactions, but still yield adequate control over client behavior by the infrastructure.

Measurement collection in Dyson supports network-wide optimizations based on both AP and client-side knowledge of the network state. This provides the CC with global information on various factors that affect client performance, such as traffic patterns, interference, hidden terminals, and congestion. This approach obviates the need for a separate wireless monitoring infrastructure [13, 7].

Each client and AP in the system records a set of statis-

Measurement	**Description**
`numPackets`	*Number of pkts received*
`totalBytes`	*Total bytes received*
`totalRSSI`	*Total RSSI of received pkts*
`connectivity[]`	*List of tuples* $\langle srcmac, numPkts, totalRSSI \rangle$
`packetsPerPhyRate[]`	*One counter for each PHY rate*
`totalAirtime`	*Airtime used by packets (size* $\times$ *PHY rate)*
`numTxFailures`	*Number of Tx failures*
`numRetransmissions`	*Number of ARQ retransmissions*
`airtimeUtil`	*Channel airtime utilization*

Table 1: **Measurements collected by Dyson nodes.**

tics, summarized in Table 2.1. For each received packet, a set of counters are incremented to track the total number of packets, total packet size, total airtime utilization, and other measures. Dividing counters by the number of received packets can be used to calculate mean values over a measurement window. Clients maintain a single set of these counters, whereas the AP maintains these counters on a per-associated-client basis, allowing measurements to be collected for each separate uplink. In addition to the these statistics, nodes also record the mean airtime utilization (reported by the radio hardware) of the radio channel.

APs periodically query their associated clients to collect their measurements, after which clients reset their counters. The AP then pushes the collected client measurements, as well as its own, to the CC. The AP's measurement collection period can be adjusted by the CC to tradeoff reporting latency and measurement traffic overhead. Our measurements in Section 4.8 show that for moderate-sized networks, this overhead is less than 1%.

2.2 Network map

The central controller uses collected measurements to maintain a *network map* representing the global state of the Dyson network. The network map is the key data structure accessed by Dyson's policies (Section 4) in order to drive reconfiguration. The network map is updated each time new measurements are pushed to the CC by an AP. Policies can read the complete network map and push new information into the network map. This allows individual policies to augment the global state maintained by the CC, as well as enabling policies to be composed.

The map consists of several components:

Node location: A table of the physical location of each AP and client in the system, indexed by MAC address. AP locations are static, whereas client locations are computed using the algorithm described in [12]. This information can be used for determining the physical location of network hotspots, and by policies that consider client mobility.

Connectivity: A directed connectivity graph is main-

`SetRate`(r)	Set PHY rate
`SetChannel`(c)	Set channel
`SetTxLevel`(t)	Set transmission power level
`SetCCAThresh`(t)	Set CCA threshold
`SetPriority`(p)	Set 802.11e priority
`Throttle`(r)	Throttle outgoing traffic at the specified rate r
`Handoff` ($c, ap, chan$)	Handoff client c to AP ap on channel $chan$
`AcceptClient` (c)	Associate AP with client c
`EjectClient` (c)	Disassociate client c

Table 2: **The Dyson command API. Commands in bold are applicable to APs only.**

tained, where vertices represent nodes (clients or APs) and edges represent the ability of one node to overhear packets of another node. For each unique MAC address that a node overhears during a measurement interval, the mean RSSI value of packets from that MAC address are reported to the CC. The connectivity graph contains a directed edge for each pair of MAC addresses. While clients are only capable of reporting links on their current channel, APs can use a secondary radio to perform background scanning and report observed connectivity on every channel. An edge is removed from the graph if no packets are observed on the link for 30 seconds. The connectivity graph is used in client-AP associations, detecting hidden terminals, and managing handoffs.

Airtime utilization: Each node measures the airtime utilization of the radio channel in its vicinity. The network map includes a hash table mapping a node's MAC address and channel number to its airtime utilization estimate. This information can be used by a wide range of policies to detect congestion, balance uplink and downlink fairness, and optimize client/AP associations. APs can measure airtime on every channel using the secondary scanning radio.

Historical measurements: Collected measurements are also stored in a persistent database, permitting policies to make use of historical information when making decisions about network reconfiguration. As an example, a policy may wish to consider the historical interference pattern between two APs, or variance in the network congestion at different hours of the day, when driving network reconfigurations.

The network map serves primarily as input to the various policies for driving network configurations. However, it can also serve an auxiliary role to assist a network administrator in understanding AP coverage and sources of performance degradation. For example, visualizing the airtime utilization graph as well as the associated client and AP locations can provide real-time information on network hotspots.

2.3 Central controller

The central controller is responsible for managing the entire Dyson network based on collected measurements from clients and APs. Its job is to apply administrator-defined *policies* to the current network map, and issue *commands* to set parameters of clients and APs according to the policy decisions.

The Dyson command API is shown in Table 2. These commands are intended to provide a rich set of knobs for controlling the network's operation while limiting overheads for command issue. Commands set parameters such as the transmission power level, CCA threshold, 802.11e priority levels, and PHY data rate. The `Handoff`, `AcceptClient`, and `EjectClient` commands control client-AP associations, as described in the next section. Note that clients do not decide themselves which AP to associate with; this is under the control of the Dyson infrastructure.

The CC sends commands to APs directly. Commands to clients are relayed via the AP that the client is currently associated with; in this way the client need not be aware of the CC's identity, and the CC's functionality can be decentralized. Commands are exchanged using MAC-layer control messages which are ACKed by the receiving node. For AP-client commands, ARQ is used to ensure commands are delivered reliably.

Support for legacy clients: Dyson can support legacy 802.11 clients without the extensions described above. Of course, this implies reduced functionality as it is not possible to directly obtain client-side measurements, nor control many aspects of client operation. The CC is able to control client-AP associations for legacy clients, giving the infrastructure control over at least which APs and channels those clients occupy. AP-side measurements can account for any associated legacy clients allowing the system to have visibility into the impact of legacy client traffic. Dyson policies use a reduced control API (that only contains the relevant calls) to interact with legacy clients.

2.4 Policy Engine

Dyson's architecture is designed to support extensibility, composability, and separation of concerns, in order to tune network performance as well as impose site- and client-specific policies. Each policy is encapsulated in a software module that runs on the CC, takes the network map as input, and issues commands to APs and clients as output. As described above, policies can also update and augment the network map itself.

Dyson has a predefined set of policy modules providing commonly-used functionality, but it is possible for new policies to be implemented and loaded into the central controller as needed. Policies are implemented in Python

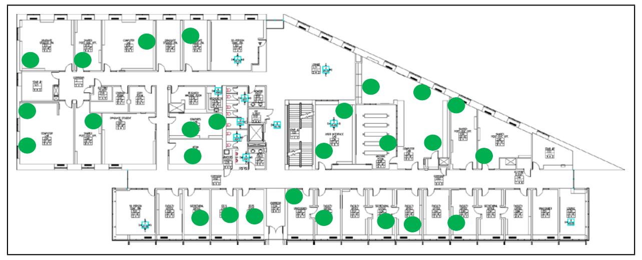


Figure 2: **Dyson testbed deployment**

and are relatively easy to write, as we will show below. This approach enables network designers to update the policies used by a Dyson network installation over time in response to new demands or shifting priorities. We also envision third parties developing new policies for Dyson that can be readily plugged into an existing deployment.

In our current design, policy composition and dependencies must be handled manually by policy designers. There is nothing to prevent two policies from "competing" (say, by issuing conflicting commands in response to the same event in the network); each policy should clearly document its own behavior to avoid unexpected results.

Each policy runs as a separate thread on the CC and is responsible for its own scheduling. Typically, a policy will run with some predefined period, but a policy can also trigger execution on some condition being met (for example, an update to some element in the network map). Standard thread synchronization primitives can be used to implement more sophisticated cross-policy interactions.

In Section 4, we demonstrate a set of policies that highlight different aspects of Dyson's global network visibility and deep control over both APs and clients.

3 Implementation and Testbed

We have implemented a prototype of the Dyson architecture using the ALIX 2c2 single-board computer (500 MHz AMD Geode processor with 256 MB DRAM) running FreeBSD 7, coupled with dual CM 9 Atheros-based 802.11a/b/g radios. Each node can act as either a Dyson client or an AP; only APs make use of the second radio for collecting channel utilization measurements.

We have deployed a testbed of 28 nodes across one floor of an academic office building, as shown in Figure 2. Each node is connected to an Ethernet network for control. The central controller is implemented on a separate machine running FreeBSD with 2 GB of RAM. All experiments presented in this paper use 802.11a to avoid interference with existing 802.11b/g networks in the building.

To support Dyson, we modified the FreeBSD Atheros driver to add support for statistics collection and the Dyson command API, as well as to disable local rate adaptation. Each node runs a Python-based daemon that exposes the Dyson measurements and command API via an XML-RPC interface, and communicates with the modified Atheros driver through *ioctl* calls. The central controller is also implemented in Python; policies are loaded as Python modules at startup time.

The commands listed in Table 2 were implemented via modifications to the Atheros driver. Most of the commands (such as `SetTxLevel`, `SetChannel`, and so forth) simply set driver parameters. `Handoff` informs a client to switch channels and associate with the specified AP. This eliminates the need for scanning, provided the destination AP is still on the specified channel. The `Throttle` command makes use of *dummynet*, a FreeBSD traffic shaping tool, to limit the rate of outgoing traffic. `Throttle` simply sets the dummynet outgoing bandwidth limit on the radio interface to the specific rate.

4 Policies and Evaluation

The primary goal of this section is to demonstrate that the *extensibility* afforded by the Dyson architecture is both desirable and feasible. To do so, we focus on five interrelated issues.

First, we show that the Dyson architecture enables interesting, non-trivial customizations of WLAN deployment that either improve performance, or enable new features. This is our key contribution.

Second, we show that the customizations are easy to realize. Unless Dyson makes it easy to customize WLANs, the fact that it enables interesting customizations is of little practical value. Also, demonstrating the ease of customization reaffirms our programming model, and validates our contention that our chosen API provides the right level of control for our purposes.

To address these two issues, we demonstrate a set of four policies that customize WLAN deployments in a variety of ways. We show how these policies can be realized via simple Python scripts, and illustrate how our APIs provide the right level of abstraction to achieve this.

Third, we discuss how multiple policies co-exist within the Dyson framework. We will show how Dyson allows different policies to be run in different parts of the network. Dyson requires policies to document their behavior and it lets the system administrator decide which polices can run safely together.

Fourth, we show that Dyson can operate at a sufficiently large scale to be of practical use. We demonstrate this via large-scale experiments with one of our policies, and also by careful micro-benchmarking of several aspects of the Dyson architecture.

Fifth, we show that although Dyson can operate without client-side modifications, using Dyson-enabled clients significantly improves performance, and enables features

```
# Input: client MAC, list of (AP MAC, RSSI) for
#   each received probe request
# Output: client MAC, AP with highest
#   available capacity
def (clientmac, heard_list):
  global ap_list, ap_list_lock, ratemap
  best_ap = None
  max_ac = -1

  # Compute available capacity for each AP
  # Pick AP with the highest value
  for (apmac, rssi) in heard_list:
    ap_list_lock.acquire()
    data_rate = ratemap.get_rate(rssi)
    airtime = ap_list[apmac].airtime
    avail_capacity = data_rate * (1.0 - airtime)
    if avail_capacity > max_ac:
      max_ac = avail_capacity
      best_ap = ap_list[apmac]
    ap_list_lock.release()

  # Assign channel if no clients already
  if (best_ap.channel == -1):
    best_ap.assign_channel()
  # Associate client
  best_ap.AcceptClient(clientmac)

def run(self):
  global pending_associations
  global pending_associations_lock

  while (True):
    pending_associations_lock.acquire()
    map(compute_ac, pending_associations)
    pending_associations = []
    pending_associations_lock.release()
    time.sleep(5)
```

Figure 3: **The Dyson capacity-aware association policy.**

that would not otherwise be possible. This is essential because client modifications are generally considered to be disruptive and expensive. We illustrate this by running one of the policies both with and without client modifications.

4.1 Customizing associations

We begin with a simple demonstration of Dyson's extensibility mechanisms at work. In Figure 3, we present a policy that associates clients with access points based on the estimated *channel capacity* at each AP. This policy is similar to the one proposed in DenseAP [22], but rather than being implemented as a complete, standalone system, using Dyson we implement it in approximately 40 lines of Python code.

The key idea is to use information on airtime utilization and an estimate of the feasible PHY rates to determine the best AP with which to associate a given client. The policy runs every 5 seconds. On each iteration, it scans over a list of probe requests received from clients. A given probe request may have been overheard by multiple APs. For each AP, the available channel capacity is computed, which is the product of the estimated PHY rate at which the client and AP will communicate, and the inverse of the AP's measured airtime utilization. The PHY rate is determined using a *rate map* that maps the RSSI of the received

```
  # Compute available capacity for each AP
  # Pick AP with the highest value. But ensure
  # this association is not an exception
  for (apmac, rssi) in heard_list:
    ap_list_lock.acquire()
    data_rate = ratemap.get_rate(rssi)
    airtime = ap_list[apmac].airtime
    avail_capacity = data_rate * (1.0 - airtime)

    #Check if this association is prohibited
    # (Code not shown ... )
    if is_exception(clientmac, apmac):
       if avail_capacity > max_ac:
         max_ac = avail_capacity
         best_ap = ap_list[apmac]
    ap_list_lock.release()
```

Figure 4: **A snippet of modifications necessary to the capacity-aware association to account for interference.**

probe request to the maximum feasible PHY rate for that client/AP pair. The rate map computation is performed separately and is not shown in the code in Figure 3.

The AP with the maximum available capacity is selected as the one that the client should associate with. If the AP currently has no clients, a channel is assigned to it, and the AP is then instructed to accept the client's probe request, by sending a probe response. This policy is used as the default association policy in Dyson and is used by the subsequent policies unless otherwise specified. We have performed experiments to confirm that its performance is similar to DenseAP's association scheme [22]. *The key takeaway from this example is the ease and conciseness of writing a Dyson policy whose functionality mimics that of a system proposed earlier.*

4.2 Interference-aware association policy

The association policy described in the previous section does not explicitly take interference into account. Recent research [26] has shown the benefits of explicitly accounting for interference between clients. A system administrator may wish to utilize such knowledge to improve associations in the WLAN. Prior systems [26, 22] do not permit such rapid changes to the WLAN.

However, due to the flexibility of Dyson, as seen in Figure 4, we can easily modify the basic policy shown in Figure 3 to account for interference. The change to the policy is minor because it checks if a particular client associating with an AP is part of the same exception.

A simple case of interference is when two clients, associated with different APs, can hear each other. The Dyson central controller can easily detect such cases (see Figure 5, based on information reported by clients and can take remedial action if necessary. For example, it can change the channel of one of the APs.

The interference-aware association policy periodically scans the global connectivity graph and detects cases in which two APs and two clients form an interference relationship similar to that in Figure 5. The policy changes the

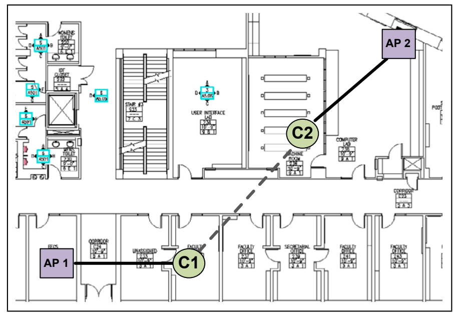


Figure 5: **Interference example. The two clients determine they interfere with each other, despite being associated with different APs.**

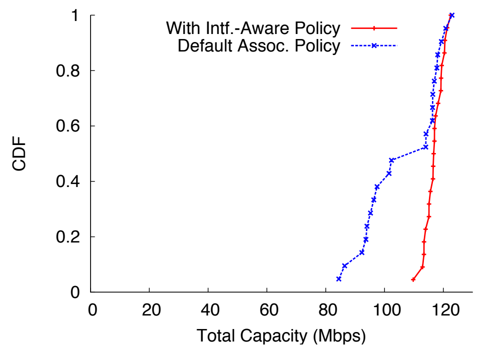


Figure 6: **Impact of interference mitigation policy on throughput of nodes across the entire floor across 20 separate runs.**

channel of the particular AP (and its clients) with fewer associated clients. The affected nodes are informed of the channel switch directly via a command, thereby avoiding the overhead of re-discovery and re-association if the policy were to simply change the channel of the AP. Note that this simple greedy algorithm might induce a new interference condition elsewhere in the network, necessitating another channel switch. To avoid oscillations, we do not change an AP's channel more than once every 10 minutes.

To demonstrate the impact of this policy, we ran an experiment with 5 of the testbed nodes acting as APs and 16 nodes acting as clients. The APs and clients were distributed roughly evenly across the testbed. The clients generated uplink traffic using greedy TCP flows for 5 minutes. We first ran the experiment using the association policy described in Figure 3, and then repeated the experiment using the modification described in Figure 4. We repeated the entire experiment 20 times.

For each run, we obtained the total capacity of the network by summing up the total throughput achieved via each AP. The results, as seen in Figure 6 show that the interference aware association policy significantly improves the capacity of the network.

There are two key takeaways from this example. First, we show how interesting performance optimizations can be enabled in Dyson with just a few lines of Python code. Second, we show the usefulness of modifying clients:

```
#Returns clients whose airtime reservations
#have not been met
def cull_special_clients(clients):
  resv_clients = []
  resv_count = 0
  for c in clients:
    if is_reserved(c)
      resv_count = resv_count + 1
      if c.get_airtime() < c.res_airtime:
        resv_clients[] = c
  return (resv_clients, resv_count)

#Throttle other clients as necessary
def throttle (ap, c,f):
  for client in ap.clients:
    if client.mac != c.mac and
       !is_reserved(c):

       #throttle/de-throttle by f%
       throttle(client.mac, f)

#Returns residual airtime at ap
def get_residual_at(ap):
# (Code not shown ... )

def run(self):
  global ap_list, ap_list_lock
  global client_list, client_list_lock
  while (True):
    ap_list_lock.acquire()
    for ap in apmap:
     residual_at = get_residual_at(ap)
     (res_clients, resv_count) =
           cull_special_clients(ap.clients,
                                residual_at)

     if len(l) > 0:
      #For each special client, throttle other
      #associated APs until targets are met
      for c in res_clients: throttle(ap, c, F)
     elif resv_count > 0:
      #needs of special client are met
      #de-throttle other clients
      for c in res_clients: throttle(ap, c, -F)
    ap_list_lock.release()
```

Figure 7: **The air-time reservation policy**

without detailed measurements from the clients, it would not have been possible to identify interfering pairs.

A more complex version of this policy can take historical knowledge of the network into account. For example, once an interference pattern between locations is determined, the system can proactively assign APs and clients in those locations to different channels.

4.3 User-specific airtime reservation

We now demonstrate that the Dyson architecture can enable new functionality that is not available in traditional WLAN systems, namely, reserving airtime for a specific user or group of users. Note, while some Wi-Fi networks do enable 802.11e for prioritization, 11e lacks the ability to reserve a certain fraction of airtime for a given station.

The network designer can easily accomplish this task with Dyson using the policy shown in Figure 7. A high-priority client c_h is identified by its MAC address. For all other clients $\{c_1, c_2, ...c_k\}$ associated with the same AP, the residual airtime $R = 1 - \sum_i ATU(c_i)$ is computed.

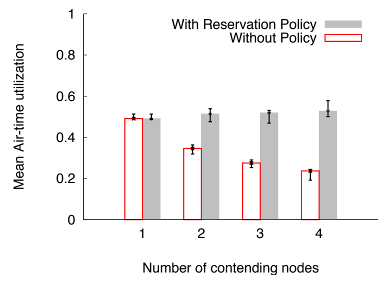
If R is less than the target airtime for c_h, the policy iterates through the list of low-priority clients, and *throttles* each of their transmission rates by a fraction f of their current throughput. This is performed using the Dyson `Throttle` command, shown in Table 2. Throttling is performed periodically until the residual airtime exceeds the target. On the other hand, if there are special users and their needs are being met, the policy then attempts to de-throttle other clients which may have been throttled.

This approach makes no assumptions about the nature of client traffic, and simply "searches" for the throttle setpoints that yield adequate airtime to the high-priority client. It is also conservative in the sense that clients are throttled equally, without regards to their load. A straightforward enhancement would throttle higher-load clients first. Note that when c_h disassociates with the AP, the low-priority clients are unthrottled; likewise, when a client moves to another AP is throttle is released. Multiple high-priority clients can also be supported on a single AP as long as their airtime targets do not exceed 100%; in that case, each high-priority client receives a weighted proportional share of the airtime. Note that this policy requires the ability to control clients directly, and hence is not possible to implement without client modifications.

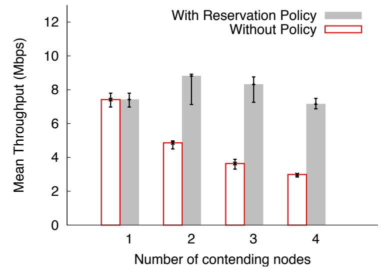
We demonstrate this policy using the following experiment. The setup consists of four APs and 11 clients. One of the clients is given an airtime reservation of 50%. For this experiment, we manually set the APs to different channels, and associate one non-privileged client with AP1, two non-privileged clients with AP2, and so on. The privileged client is nomadic. It associates with each of the four APs in turn for 10 minutes each. All clients download data as fast as they can using *iperf* UDP flows. We first perform the experiment without any reservation policy, and then repeat it after reserving 50% of the airtime for the privileged user. We repeat the entire experiment 10 times for statistical significance.

The impact of the policy is shown in Figure 4.3. In the absence of the reservation policy, the fraction of airtime received by the privileged user drops as the number of non-privileged clients increases. However, when the reservation policy is in force, the privileged user always receives the 50% reserved fraction of the airtime.

As a side note, remember that providing *guaranteed* airtime does not translate to guaranteed throughput, because of the variability in radio link quality of the link between the privileged client and each of the APs. The throughput received by the privileged client in the previous experiment is shown in Figure 8(b). Even though the privileged client receives a fixed amount of airtime, the throughput it achieves varies for different APs. We can easily modify the policy described above to ensure that the reserved airtime varies in inverse proportion to the quality of the channel seen by the privileged user, to ensure that



(a) Impact on airtime



(b) Impact on throughput

Figure 8: **Impact of airtime reservation policy on the airtime and throughput received by a single privileged user competing with several other clients. Error bars represent 10th and 90th percentiles.**

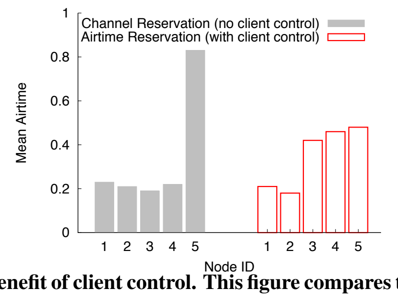
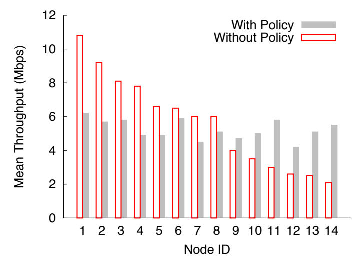


Figure 9: **Benefit of client control. This figure compares two different policies: one that exercises control over clients (airtime reservation) to one that only controls APs (channel reservation) only. Node 5 is the special user guaranteed at least 50% of the airtime. Client-side throttling leads to a more fair distribution of airtime across clients than reserving channels alone.**

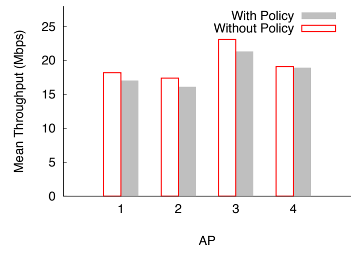
the bulk transfers done by the user receive certain guaranteed target throughput. Though we have implemented and tested this policy, we elide details due to lack of space.

There are two key takeaways from this example. First, we show that Dyson can enable novel functionality, that is not otherwise available. Second, this example also demonstrates that certain kinds of functionality can only be enabled via client modifications.

Benefits of client-side control: Throttling airtime usage at the client gives Dyson direct control over client behavior. The question that arises is whether such client control is strictly necessary. As an alternative, consider a policy that does not assume any client extensions. As opposed to reserving airtime, the policy reserves an entire channel on a given AP for special users, requiring changes to client-AP associations in order to avoid inter-



(a) Throughput for various nodes with/without load balancing



(b) Total throughput at each AP with/without load balancing

Figure 10: **Large-scale load balancing experiment.**

ference between special users and regular users. We have implemented such a policy and compared it against the airtime reservation policy. The setup consists of two APs and four clients (two associated with each AP) performing uplink TCP traffic. A fifth client arrives, one of which is guaranteed 50% of the airtime. In the channel reservation policy, clients from one AP are moved to the other, which has four clients contending. The fifth user is given exclusive access to the vacated AP. With the airtime reservation policy, the fifth user associates with one of the APs where the other clients are throttled (the clients at the other AP are left untouched). The results are shown in Figure 4.3, which shows that the channel reservation policy leads to a less fair distribution of airtime than the airtime reservation policy. This experiment demonstrates the added value of control over both clients and the infrastructure. Note, another possible approach here would have been to enable 802.11e priority queues. However, our intention is to demonstrate the ease with which different kinds of policies, each offering a different degree of control, can be implemented and deployed in Dyson.

4.4 Uplink/downlink load balancing

In this section, we show how Dyson can be used to correct a basic flaw in the 802.11 architecture, called the upload-download anomaly [24]. The 802.11 MAC ensures that each node with pending packets gets equal opportunity to access the channel. Consider a WLAN with ten clients associated with a single AP. Nine of these clients download data from the network, while one client uploads data to the network. Because the AP and the upload client are the only two nodes that have pending data to send, they share the airtime equally. The upload client gets to transmit roughly half the time, while the nine download clients *together* share the remaining time!

Although traditionally the majority of the traffic in WLANs has been download traffic [4], this pattern is expected to change as WLANs become more popular as access networks. As the above example shows, only a few upload clients are needed to cause significant unfairness.

In a Dyson-enabled network, we can easily address this problem via the following simple policy. The policy attempts to balance the total volume of uplink and downlink traffic handled by an AP. For each AP, associated clients are classified as either predominantly upload or download, based on the ratio of their throughput in each direction. We then compute the ratio of the mean throughput for upload and download clients. If the ratio exceeds a specified threshold, it suggests that upload clients are dominant and that rebalancing is required for this AP.

As a simple approach, the policy throttles upload clients in an attempt to bring the upload/download ratio closer to 1. Upload clients are ordered by decreasing uplink throughput, and the "heaviest" upload client is throttled to 50% of its current throughput. The policy then sleeps for 10 sec and re-evaluates the upload/download ratio, iteratively throttling the highest-throughput upload client until the ratio between the mean upload and mean download throughput at the AP falls to less than 1.5.

This policy relies on client cooperation to solve the problem. This is not strictly necessary, but other remedies are more disruptive. For example, the policy could attempt to modify client-AP associations to balance the number of download clients across APs. However, such a policy is not always guaranteed to achieve the correct distribution of clients required. The client-throttling approach described above is much simpler to implement.

To demonstrate this policy, we performed an experiment with 4 APs and 15 client nodes. Client-AP associations were determined using the capacity-aware association policy (Section 4.1). Note that different APs have a different number of associated clients. Each AP is assigned to a different channel by the policy.

One client associated with each AP generates upload traffic, while others generate download traffic. We ran the experiment twice, first without the uplink/downlink load balancing policy running, and then with the policy enabled. Figure 10(a) shows the distribution of the throughput obtained by each of the clients with and without the policy running. There is a clear bandwidth inequity in the default case, but the policy produces a much more balanced distribution of network capacity to each client.

Of course, achieving fairness is often at odds with maximizing overall network capacity. Figure 10(b) shows the aggregate throughput at each AP before and after the policy was enabled. As the figure shows, there is a slight dip in overall bandwidth usage at each AP: 5.7% on average.

This example illustrates that a simple Dyson policy can correct problems inherent in the 802.11 architecture by using feedback from the client and by exercising control over clients.

4.5 VoIP-aware handoffs

As a another example of Dyson's ability to enable network-wide optimizations, we present an example policy that assigns VoIP clients to a different set of APs than other clients, to increase overall VoIP call capacity and avoid bulk transfers from impacting VoIP call quality. This policy assumes that clients have been classified as VoIP or non-VoIP clients, for example, based on the client's MAC address (e.g., for WiFi VoIP handsets). Due to lack of space, we omit the python code for this policy.

For each VoIP client that is assigned to a non-VoIP AP, the policy identifies a new VoIP-specific AP with which to associate. For each VoIP AP that the client can potentially connect to (based on the connectivity graph), the available capacity metric is computed, as described earlier. The client is simply handed off to the VoIP AP with the highest available capacity.

Although there are more sophisticated techniques to improve VoIP capacity in WiFi networks [29], this policy is simply intended to demonstrate Dyson's interfaces and programmability. This simple policy can be extended in various ways. For example, the assignment of APs as VoIP or non-VoIP (which is currently static) can be performed in a dynamic fashion based on VoIP call load. Likewise, the number of VoIP clients assigned to each AP could be taken into consideration. We elide the details due to lack of space.

We carry out the following experiment. We configured two nodes near each other as APs, and another four nodes as clients. The capacity-aware association policy described in Section 4.1 was used, resulting in two clients being associated with each AP. The APs were assigned to different channels by the association policy.

Two clients, on separate APs, initiated a bidirectional VoIP flow while the other two clients began large saturating download traffic using *iperf*. The VoIP flows each use a standard g729 VoIP codec that generates 50-byte packets at a rate of 31.2 Kbps.

The bulk flows adversely affect the VoIP flows in terms of introducing increased packet jitter, which causes the quality of the VoIP call to degrade. A common requirement for VoIP calls is that jitter should be no greater than 2ms [3]. Figure 4.5 shows that with the default configuration, up to 2.17 ms of jitter is induced by the bulk flows on each VoIP call. Note, this is done with a few clients.

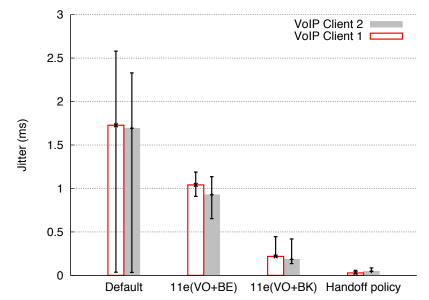


Figure 11: **Effect of 802.11e prioritization and VoIP-aware handoffs on VoIP jitter.** ***This is an experiment with two VoIP clients competing with two bulk-download clients, with two APs on different channels. Using the*** **default** ***policy, one VoIP client and one bulk client are assigned to each AP. The*** **11e(VO+BE)** ***policy uses 802.11e prioritization, assigning bulk clients to the*** **best effort** ***queue.*** **11e(VO+BK)** ***assigns bulk clients to the*** **background** ***queue.***

Next, we enabled the VoIP handoff policy, which migrates VoIP clients to one of the APs and the bulk flows to the other. As Figure 4.5 shows, this substantially reduces the jitter to a mean of 0.02 ms. This also causes the bulk transfers to share the channel on a single AP, causing their throughputs to degrade; prior to migration, each bulk flow obtained 24 Mbps of throughput. After migration, each bulk flow degrades to 12 Mbps. This is an explicit tradeoff between providing good service to VoIP clients versus the (arguably less severe) impact on bulk flows.

As an alternative, we also experimented with using 802.11e priority levels, with a simple policy that uses the `SetPriority` command. We set up one experiment in which the VoIP clients were configured to use the 802.11 *voice* priority and the bulk clients to use the 802.11e *best effort* priority, while maintaining the original AP associations. Another experiment uses the 802.11e *background* priority, which is lower than best-effort. As the figure shows, 802.11e priorities do mitigate some of the jitter effects, but do not operate as well as the handoff policy. Each bulk client received 24 Mbps of throughput using the best-effort priority, and 18 MBps using the background priority. In general, it will not always be possible to cleanly separate VoIP clients from others in the network, so in general a combination of migration (where possible) and 802.11e priority levels is likely to be the most effective solution.

Note this policy could have been implemented in prior systems, such as SMARTA [6], MDG [11], or DenseAP [22]. Note, however, that this is easy to do so in Dyson via the exposed API. The *key takeaway from this example is the platform Dyson provides to develop and deploy such polices very quickly.*

4.6 Running multiple policies together

So far, we have demonstrated each policy in isolation. However, a network administrator will often want to run

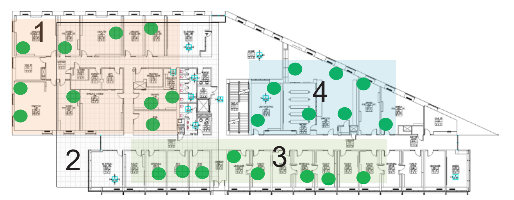


Figure 12: **The floor map for our testbed, which was divided into four regions. Each region is configured to run a different set of policies.**

multiple policies simultaneously and compose their behavior. For example, different types of traffic may need to be given different priorities in different parts of the building, or at different times of day.

Dyson supports running multiple policies on different spatial regions of the network, and varying the set of policies that are active over time. Furthermore, Dyson can track client locations, and ensure that appropriate polices are applied depending on the client's location.

To illustrate the use of multiple policies, we ran the following experiment on our testbed. We divide our floor into four regions as shown in Figure 4.6. Regions 1, 3, and 4 do not overlap with each other, whereas region 2 overlaps with 1 and 3. For each region, we configure Dyson to run the following policies.

Region 1: We reserve one of the APs (and one of the channels) for VoIP traffic and enable the VoIP sifting policy described in Section 4.5. This ensures we dedicate resources to VoIP clients in this area.

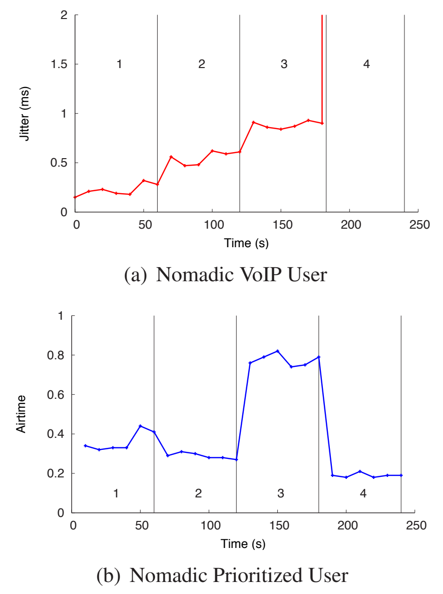
Region 2: In this region, we run the interference-aware association policy described in Section 4.2.

Region 3: In this region, we reserve 80% of the airtime for a set of users. For example, the administrator may want to deploy such a policy in a region where faculty offices are located, giving faculty members preferential treatment.

Region 4: In this region, we disable all VoIP calls using a policy that dissociates any client that is transmitting VoIP traffic.

Each region contains at least two APs, and each AP has anywhere between 2–4 clients associated with them performing variable bit rate UDP traffic. In addition, there are two nomadic users, whose behavior we monitor. User 1 is a VoIP client who starts walking in region 1, 2, 3, and finally ends in 4. User 2 is another nomadic user who is performing bulk TCP transfers. This user is a faculty member who is guaranteed by the policy in Region 3 to get 80% of the airtime. Each user spends approximately 60 seconds in each location.

The jitter for User 1 and airtime for User 2 are shown in Figure 4.6. As the figure shows, the VoIP user experiences significantly less jitter when she is in Region 1, compared to other regions, because an AP and a channel are reserved for VoIP calls in this region. Furthermore, when she enters Region 4, her service is cut off. We also



(a) Nomadic VoIP User

(b) Nomadic Prioritized User

Figure 13: **Time series graphs for the two different nomadic users as they walk through the various regions.**

see that User 2 gets her reserved airtime only when she is in Region 3, as expected.

The key takeaway from this example is that Dyson can successfully run multiple policies in a single network and apply policies in a location-specific manner.

This example does not address how multiple polices may interact with each other. In our current implementation, we rely on the network administrator to determine if policies may have adverse interactions or may simply cancel out each other's decisions. It is assumed that policies themselves are well-documented and that the administrator can reason about the possible interactions between multiple policies running on the same parts of the network. In our future work, we plan to build tools to help administrators detect and resolve such conflicts.

4.7 Other Policies

Due to a lack of space, we are unable to present results from other policies we have implemented and experimented with, in the Dyson system. One such policiy is a one that reduces the cumulative handoffs for certain mobility paths. Dyson can use historical knowledge of client mobility patterns to optimize AP handoffs. Since mobile handoffs are expensive and can lead to temporary connectivity loss, it is important to avoid redundant or poorly-chosen handoffs. The key idea is to predict the next AP a client will encounter while roaming, in order to avoid handing off to a different AP that will quickly go out of range. This is possible, since in many workplaces, users are more likely to travel along certain paths than others. We built and deployed this policy on our floorwide testbed and we found, on average it halves the number of handoffs

Stats interval (s)	10th	Median	90th
1	3.5%	4.3%	9.3%
5	0.8%	3.9%	6.1%
10	0.0%	3.5%	6.2%

Table 3: **CPU utilization at an AP with eight clients measured over a period of 10 minutes, for various statistics reporting intervals.**

for well traversed paths.

Along the lines of the airtime reservation policy, we have also implemented a bandwidth reservation policy whereby we guarantee a certain bandwidth to a user, and we throttle other clients nearby until we reach the desired target. There are host of other policies that are variants of the policies described in this paper, that we are currently working in the Dyson system.

4.8 Microbenchmarks

In this section, we study the overheads imposed by the Dyson architecture on clients and APs and the amount of control traffic generated by Dyson nodes. We will also study the performance penalty caused by client handoffs, as Dyson uses this mechanism often.

AP and client overheads: We examine the CPU utilization of the AP. Recall that we use an ALIX 2c2 with a 500 MHz AMD Geode, and the Dyson software is implemented in Python. We configured an AP with eight clients and varied the intervals at which the AP reported statistics to the CC. As seen in Table 3 the median utilization is still low. Since clients are periodically sending statistics to the AP, we also measured the CPU utilization at a client over a period of ten minutes. We found the modified Dyson drivers added negligible overhead in terms of CPU and memory utilization ($< 1\%$).

Traffic overhead for measurments collection: We also measured the traffic sent by clients and APs to the CC. The AP's measurement collection period can be adjusted by the CC to tradeoff reporting latency and measurement traffic overhead. As an estimate of this overhead, each client measurement packet requires at most 850 bytes, including MAC headers. At the lowest OFDM PHY rate of 6 Mbps, this requires $1184\mu s$ to transmit (accounting for MAC and framing overheads). Therefore, an AP with 20 clients will require less than 1% of the radio channel for statistics collection. The AP sends all client statistics as well its own statistics to the CC. We measured the traffic sent by an AP with six clients to the CC. With a statistics reporting interval of 5 sec, the AP generates 1638 bytes/sec in traffic to the CC, which includes overheads induced by the use of XML-RPC. This is a small fraction of the backhaul wired network capacity.

Handoff overhead: We measured the time taken for a MAC-layer handoff of a client from one AP to another. We configured two APs (on different channels) and a single client, which was initially associated with AP1. The CC then issued a `Handoff` command to migrate the client to AP2. AP1 receives this command and relays it to the client who quickly switches channels and associations. This process also includes informing AP2 to permit the new association.

Step	Time(ms)
Handoff command executed	0
Message reception (at client)	0.120
Channel change	5.6
Authentication	0.159
Association	0.359
Total	6.238

Table 4: **Handoff overhead in Dyson.**

The MAC-layer handoff overhead includes the time for the command transmission to the client, the time for the client to switch channels (between to 5 to 7 ms on the Atheros chipset), and the client's reassociation with the new AP. The end-to-end delay experienced by an application may be longer, for example, due to the settling time of the spanning-tree algorithm on the wired backbone.

The results are shown in Table 4, which shows that a MAC-layer handoff requires approximately 6.2 ms in our current prototype. This process can be further optimized, as demonstrated in [25]. Also, the use of protocols such as IAPP (Inter-Access Point Protocol) at a higher layer, in which APs cache packets during a handoff and forward them to the destination AP, can mitigate the packet loss incurred during a handoff. We have not yet implemented this approach in Dyson.

Central controller scalability: Dyson uses a central controller to control the APs and the clients. This raises questions about the scalability and fault tolerance of our architecture. If necessary, fault tolerance can be achieved using standard techniques such as primary-backup. We believe that scalability is not a concern, since the processing done at the central controller is not CPU or memory intensive for most polices that we envision. In all our experiments, the load on the central controller was negligible. However, it may be the case that certain policies or certain deployments may require extensive processing capabilities at the central controller. In such a case, it may be possible to use multiple machines to act as central controllers and use standard load balancing techniques to prevent any one machine from becoming a bottleneck. We plan to study these issues in detail as part of our future work.

5 Related Work

Dyson is complementary to a broad class of prior work on improving the performance and scalability of wireless networks through new techniques at the MAC and PHY layers [28, 9]. Our focus is on the higher-level aspects of network management that can be obtained through global observation and deep control.

Dyson is inspired by the same vision that inspired projects such as OpenFlow [19] and 4D [14], where significant intelligence resides in a central controller. The central controller makes use of global knowledge to make network-wide decisions. We note that the Dyson architecture is quite compatible with the overall OpenFlow design. We are currently investigating whether some parts of Dyson functionality (especially AP controls) can be recast in the OpenFlow model.

Several commercial systems use some form of global knowledge or a central controller for managing WLAN deployments. Aruba [1] uses central controller to do network-wide channel and power management to mitigate interference, while Meru [2] uses a central controller to speed up handoffs for mobile clients. Detailed information on how these systems work is difficult to come by - the marketing literature does not reveal much. However, commercial vendors are hampered by the need to maintain backwards compatibility with existing 802.11 networks. To the best of our knowledge, no commercial system includes a client component.

Research systems such as DenseAP [22] and DIRAC [32] also propose a centralized architecture. However, both systems explicitly assume that no special software can be run on clients, and thus are limited in what they can accomplish. Centaur [26] does use some form of client modifications, along with centralized control. However, Centaur has a narrow goal: to avoid hidden and exposed terminal issues. Dyson is a much more general system. In fact, in Section 4.2, we have shown how Dyson can find and avoid certain kinds of interference and hidden terminal problems.

Several research systems use a limited form of client cooperation. In MDG [11], clients get information from APs via special fields in the Beacon packets, and the client driver uses this information to make various decisions (e.g. associations). However, the specified interface is quite limited, and is more akin to the one proposed in the 802.11k standard [15]. Similarly, [21] uses feedback from clients to enable use of partially overlapping channels, [6] uses client-cooperation via micro-probing [5] to construct a conflict graphs [23] of the network.

The Dyson architecture, on the other hand, provides a general-purpose API for managing clients and APs, and can be viewed as a generalized version of these systems.

Systems such as SoftRepeater [8] and CMAP [30] specifically focus on client cooperation to improve WLAN performance. In SoftRepeater, clients with good connections relay packets for poorly-connected clients. Similar functionality can be implemented as a policy in the Dyson framework. In CMAP, clients collaborate to build an interference map of the network, which is used to schedule transmissions. Dyson's network map is a generalized version of CMAP's interference graph.

Another interesting design point is explored in [27]. The idea is to use bare-bones APs with analog-to-digital converters such that they are oblivious to the PHY/MAC layers being used at the client. As a result, all intelligence in the network is pushed to the clients. The Dyson approach is practical, and can be deployed with off-the-shelf 802.11 hardware.

Outside of the networking space, many systems have explored the use of extensibility via add-on modules with a well-defined programmatic interface. SPIN [10] and Exokernel [16] are classic examples of opening up the operating system interface to permit greater flexibility and application-specific control. Likewise, Lance [31] provides a policy module interface to customize data collection from a wireless sensor network.

6 Discussion and Future Work

Our prototype of Dyson has shed light on several directions for future work. First, our current design assumes that Dyson-enabled clients will be able to provide periodic measurement reports regardless of their power state. Power-constrained clients such as mobile phones routinely turn off their Wi-Fi interfaces (power save mode), and hence may not always be able to collect or report these measurements. This raises the question of what the impact of intermittent measurements collection will have on efficacy of Dyson policies. If the density of non-power-constrained clients (e.g. laptops on people's desks) is sufficiently high, good measurements can still be collected. Alternatively, a separate monitoring system like DAIR [7] can be used. In some cases, the design of polices itself will have to change to deal with partial information. We are exploring these alternatives further.

We have designed Dyson primarily for enterprise networks, where clients are under the control of a central IT department and do not need incentives for running the measurement software. We have also not considered the impact of malicious users reporting false measurements or not responding to commands. These concerns are addressed partially by the fact that in most enterprise networks, WLAN users are explicitly authenticated using protocols such as 802.1x. Another interesting possibility is to identify malicious users by comparing measurement reports from different clients [18].

In the current Dyson prototype, clients perform only passive measurements. This was done for the sake of simplicity. We plan to explore the possibility of asking clients to perform *active* measurements, e.g., asking a client to transmit a series of probe packets to measure loss rate more accurately. Concerns about overhead and battery drain will likely limit how often such active measurements are carried out. In the same vein, one may also ask certain clients to relay packets for other clients [8]. We have not considered such possibilities in the current prototype.

Finally, we note that while it is easy to write new Dyson policies, it does require some expert knowledge, especially to avoid unwanted interactions between polices that run simultaneously. We do not expect an average system administrator to have the requisite skill set. We believe if Dyson is deployed in a widespread manner, a new class of experts in programmable network management will arise who will write and distribute pre-packaged policies.

7 Conclusions

We have presented Dyson, a new architecture for extensible wireless LANs. Dyson provides an extensible network architecture that evolves with new challenges and application demands. Dyson's programmable policy framework makes it easy to customize the network's operation for site-specific needs and new services. The framework also makes it easy to store historical information about network performance, and leverage it to fine-tune network parameters. By "opening up" clients for measurements collection and control, Dyson breaks down the traditional barrier between the infrastructure and its clients, offering substantial benefits for network management.

We demonstrated how Dyson can support a wide range of policies for managing associations, specialized traffic classes (such as VoIP), mitigating interference and airtime reservations for specific users. We demonstrated the benefits of these policies using our 28-node testbed.

References

[1] Enterprise solutions from aruba networks, http://www.arubanetworks.com/solutions/enterprise.php.

[2] Meru networks - virtual cell, http://www.merunetworks.com/pdf/whitepapers/.

[3] A reference guide to all things voip, http://www.voip-info.org/wiki/view/qos.

[4] Ahmed, N., Banerjee, S., Keshav, S., Mishra, A., Papagiannaki, K., and Shrivastava, V. Interference Mitigation in Wireless LANs using Speculative Scheduling . In *MobiCom* (2007).

[5] Ahmed, N., Ismail, U., Keshav, S., and Papagiannaki, D. Online Estimation of RF Interference. In *CoNEXT* (2008).

[6] Ahmed, N., and Keshav, S. SMARTA: A Self-Managing Architecture for Thin Access Points. In *CoNEXT* (2006).

[7] Bahl, P., Chandra, R., Padhye, J., Ravindranath, L., Singh, M., Wolman, A., and Zill, B. Enhancing the Security of Corporate Wi-Fi Networks Using DAIR. In *MobiSys* (2006).

[8] Bahl, V., Chandra, R., Lee, P., Misra, V., Padhye, J., Rubenstein, D., and Yu, Y. Opportunistic Use of Client Repeaters to Improve Performance of WLANs. In *CoNext* (2008).

[9] Bejerano, Y., and Bhatia, R. S. MiFi: a framework for fairness and QoS assurance in current IEEE 802.11 Networks with Multiple Access Points. In *Infocom* (2004).

[10] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Proc. the 15th SOSP (SOSP-15)* (1995).

[11] Broustis, I., Papagiannaki, K., Krishnamurthy, S. V., Faloutsos, M., and Mhatre, V. MDG: Measurement-driven Guidelines for 802.11 WLAN Design. In *MobiCom* (2007).

[12] Chandra, R., Padhye, J., Wolman, A., and Zill, B. A Location-based Management System for Enterprise Wireless LANs. In *NSDI* (2007).

[13] Cheng, Y.-C., Afanasyev, M., Verkaik, P., Benko, P., Chiang, J., Snoeren, A. C., Voelker, G. M., and Savage, S. Automated Cross-Layer Diagnosis of Enterprise Wireless Networks. In *SIGCOMM* (2007).

[14] Greenberg, A., Hjalmtysson, G., Maltz, D., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A Clean Slate 4D Approach to Network Control and Management. In *SIGCOMM CCR* (2005).

[15] IEEE. *IEEE 802.11k-2008 — Amendment 1: Radio Resource Measurement of Wireless LANs*. June 2008.

[16] Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceño, H. M., Hunt, R., Mazières, D., Pinckney, T., Grimm, R., Jannotti, J., and Mackenzie, K. Application performance and flexibility on Exokernel systems. In *Proc. the 16th SOSP (SOSP '97)* (October 1997).

[17] Ko, B.-J., Misra, V., Padhye, J., and Rubenstein, D. Distributed channel assignment in multi-radio 802.11 mesh networks. In *WCNC* (2007).

[18] Mahajan, R., Rodrig, M., Wetherall, D., and Zahorjan, J. Sustaining Cooperation in Multi-Hop Wireless Networks. In *Proc. NSDI* (2005).

[19] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks.

[20] Mishra, A., Brik, V., Banerjee, S., Srinivasan, A., and Arbaugh, W. A Client-driven Approach for Channel Management in Wireless LANs. In *Infocom* (2006).

[21] Mishra, A., Shrivastava, V., Banerjee, S., and Arbaugh, W. Partially-overlapped Channels not considered harmful. In *ACM Sigmetrics* (2006).

[22] Murty, R., Padhye, J., Chandra, R., Wolman, A., and Zill, B. Designing High-Performance Enterprise Wireless Networks. In *NSDI* (San Francisco, CA, April 2008).

[23] Padhye, J., Agarwal, S., Padmanabhan, V., Qiu, L., Rao, A., and Zill, B. Estimation of Link Interference in Static Multi-hop Wireless Networks. In *IMC* (2005).

[24] Pilosof, S., Ramjee, R., Raz, D., Shavitt, Y., , and Sinha, P. Understanding TCP fairness over Wireless LAN. In *INFOCOM* (2003).

[25] Sharma, A., and Belding, E. M. FreeMAC: Framework for Multi-Channel MAC Development on 802.11 Hardware. In *ACM SIGCOMM PRESTO* (2008).

[26] Shrivastava, V., Ahmed, N., Rayanchu, S., Banerjee, S., Keshav, S., Papagiannaki, K., and Mishra, A. CENTAUR: Realizing the Full Potential of Centralized WLANs through a Hybrid Data Path. In *MOBICOM* (2009).

[27] Singh, S. Challenges: Wide-Area wireless NETworks (WANETs). In *MOBICOM* (2008).

[28] Vasan, A., Ramjee, R., and Woo, T. ECHOS - Enhanced Capacity 802.11 Hotspots. In *Infocom* (2005).

[29] Verkaik, P., Agarwal, Y., Gupta, R., and Snoeren, A. C. SoftSpeak: Making VoIP Play Fair in Existing 802.11 Deployments. In *NSDI* (2009).

[30] Vutukuru, M., Jamieson, K., and Balakrishnan, H. Harnessing Exposed Terminals in Wireless Networks. In *NSDI* (2008).

[31] Werner-Allen, G., Dawson-Haggerty, S., and Welsh, M. Lance: Optimizing high-resolution signal collection in wireless sensor networks. In *Proc. Sensys* (2008).

[32] Zerfos, P., Zhong, G., Cheng, J., Luo, H., Lu, S., and L, J. J.-R. DIRAC: a software-based wireless router system. In *MOBICOM* (2003).

ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory

*Biplob Debnath** *Sudipta Sengupta*‡ *Jin Li*‡

‡*Microsoft Research, Redmond, WA, USA*

**University of Minnesota, Twin Cities, USA*

Abstract

Storage deduplication has received recent interest in the research community. In scenarios where the backup process has to complete within short time windows, *inline* deduplication can help to achieve higher backup throughput. In such systems, the method of identifying duplicate data, using disk-based indexes on chunk hashes, can create throughput bottlenecks due to disk I/Os involved in index lookups. RAM prefetching and bloom-filter based techniques used by Zhu et al. [42] can avoid disk I/Os on close to 99% of the index lookups. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 1000 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash-based *index*, thereby, increasing inline deduplication throughput. Flash memory can reduce the huge gap between RAM and hard disk in terms of both cost and access times and is a suitable choice for this application.

To this end, we design a *flash-assisted* inline deduplication system using ChunkStash[1], a chunk metadata store on flash. ChunkStash uses one flash read per chunk lookup and works in concert with RAM prefetching strategies. It organizes chunk metadata in a log-structure on flash to exploit fast sequential writes. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores (2-byte) compact key signatures instead of full chunk-ids (20-byte SHA-1 hashes) so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality. Evaluations using real-world enterprise backup datasets show that ChunkStash outperforms a hard disk index based inline deduplication system by 7x-60x on the metric of backup throughput (MB/sec).

1 Introduction

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [30, 42]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using 20-byte SHA-1 hash signatures [34] for each chunk to determine whether two chunks contain identical data [42]. In *inline* storage deduplication systems, the chunks arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunks seen so far for that storage location (dataset) instance. If there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk needs to be added to the system and its hash and metadata inserted into the index. The metadata contains information like chunk length and location and can be encoded in up to 44 bytes (as in [42, 30]). The 20-byte chunk hash (also referred to as *chunk-id*) is the *key* and the 44-byte metadata is the *value*, for a total *key-value pair* size of 64 bytes.

Because deduplication systems currently need to scale to tens of terabytes to petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations which are of the order of 10msec. Since backups need to be completed over tight windows of few hours (over nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems, hence the need for a fast index for duplicate chunk detection. The index may be used in other portions of the deduplication pipeline also. For example, a recently proposed algorithm for chunking the data stream, called *bimodal chunking* [27], requires access to the chunk index to determine whether an incoming chunk has been seen earlier or not. Thus, multiple functionalities in the deduplication pipeline can benefit from a fast chunk index.

RAM prefetching and bloom-filter based techniques used by Zhu et al. [42] can avoid disk I/Os on close to 99% of the index lookups and have been incorporated in production systems like those built by Data Domain. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash memory based *key-value store*, thereby, increasing inline deduplication throughput. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times slower. Flash stands in the

[1]stash: A secret place where something is hidden or stored.

middle between DRAM and disk also in terms of cost [29] – it is about 10x cheaper than DRAM, while about 10x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk and a suitable choice for this application.

1.1 Estimating Index Lookup Speedups using Flash Memory

We present a back-of-the-envelope calculation for decrease in average chunk index lookup time when flash memory is used as the metadata store for chunk-id lookups. This can be viewed as a necessary sanity check that we undertook before plunging into a detailed design and implementation of a flash-assisted inline storage deduplication system. We use the fundamental equation for average access time in multi-level memory architectures. Let the hit ratio in RAM be h_r. Let the lookup times in RAM, flash, and hard disk be t_r, t_f, and t_d respectively.

In the hard disk index based deduplication system described in Zhu et al. [42], prefetching of chunk index portions into RAM is shown to achieve RAM hit ratios of close to 99% on the evaluated datasets. Lookup times in RAM can be estimated at $t_r = 1\mu$sec, as validated in our system implementation (since it involves several memory accesses, each taking about 50-100nsec). Hard disk lookup times are close to $t_d = 10msec$, composed of head seek and platter rotational latency components. Hence, the average lookup time in this case can be estimated as

$$t_r + (1 - h_r) * t_d = 1\mu\text{sec} + 0.01 * 10\text{msec} = 101\mu\text{sec}$$

Now let us estimate the average lookup time when flash is used to serve index lookups that miss in RAM. Flash access times are around $t_f = 100\mu$sec, as obtained through measurement in our system. Hence, the average lookup time in a flash index based system can be estimated as

$$t_r + (1 - h_r) * t_f = 1\mu\text{sec} + 0.01 * 100\mu\text{sec} = 2\mu\text{sec}$$

This calculation shows a *potential speedup of* 50*x using flash* for serving chunk metadata lookups vs. a system that uses hard disk for the same. *That is a speedup of more than one order of magnitude.* At 50x index lookup speedups, other parts of the system could become bottlenecks, e.g., operating system and network/disk bottlenecks for data transfer. So we do not expect the overall system speedup (in terms of backup throughput MB/sec) to be 50x in a real implementation. However, the point we want to drive home here is that flash memory technology can help to get the index lookup portion of inline storage deduplication systems far out on the scaling curve.

1.2 Flash Memory and Our Design

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have led to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, and cameras.

However, it is only recently that flash memory, in the form of Solid State Drives (SSDs), is seeing widespread adoption in desktop and server applications. For example, MySpace.com recently switched from using hard disk drives in its servers to using PCI Express (PCIe) cards loaded with solid state flash chips as primary storage for its data center operations [6]. Also very recently, Facebook announced the release of Flashcache, a simple write back persistent block cache designed to accelerate reads and writes from slower rotational media (hard disks) by caching data in SSDs [7]. These applications have different storage access patterns than typical consumer devices and pose new challenges to flash media to deliver sustained and high throughput (and low latency).

These challenges arising from new applications of flash are being addressed at different layers of the storage stack by flash device vendors and system builders, with the former focusing on techniques at the device driver software level and inside the device, and the latter driving innovation at the operating system and application layers. The work in this paper falls in the latter category. To get the maximum performance per dollar out of SSDs, it is necessary to use flash aware data structures and algorithms that work around constraints of flash media (e.g., avoid or reduce small random writes that not only have a higher latency but also reduce flash device lifetimes through increased page wearing).

To this end, we present the design and evaluation of ChunkStash, a *flash-assisted* inline storage deduplication system incorporating a high performance chunk metadata store on flash. When a key-value pair (i.e., chunk-id and its metadata) is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index employing a variant of cuckoo hashing [35] and compact key signatures is used to index the chunk metadata stored in flash memory and serve chunk-id lookups using one flash read per lookup. ChunkStash works in concert with existing RAM prefetching strategies. The flash requirements of ChunkStash are well within the range of currently available SSD capacities – as an example, ChunkStash can index order of *terabytes* of unique (deduplicated) data using order of *tens of gigabytes* of flash. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality.

In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe the internal architecture of SSDs in Section 2.

1.3 Our Contributions

The contributions of this paper are summarized as follows:

- **Chunk metadata store on flash:** ChunkStash organizes key-value pairs (corresponding to chunk-id and metadata) in a log-structured manner on flash to exploit fast sequential write property of flash device. It serves lookups on chunk-ids (20-byte SHA-1 hash) using one flash read per lookup.

- **Specialized space efficient RAM hash table index:** ChunkStash uses an in-memory hash table to index key-value pairs on flash, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality.

- **Evaluation on enterprise datasets:** We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [42]. For the hard disk index based system, we use BerkeleyDB [1], an embedded key-value store application that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include "a hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms. We use three enterprise backup datasets (two full backups for each) to drive and evaluate the design of ChunkStash. Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 7x-60x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-4x.

The rest of the paper is organized as follows. We provide an overview of flash-based SSD in Section 2. We develop the design of ChunkStash in Section 3. We evaluate ChunkStash on enterprise datasets and compare it with our implementation of a hard disk index based inline deduplication system in Section 4. We review related work in Section 5. Finally, we conclude in Section 6.

2 Flash-based SSD

A Solid State Drive(SSD) consists of flash chip(s) and flash translation layer (FTL). In a flash chip, data is stored in an array of flash memory blocks. Each block spans 32-64 pages, where a page is the smallest unit of read and write operations. In flash memory, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, unlike disk, read and write operations do not exhibit symmetric behavior. This asymmetry arises as flash memory does not allow in-place update (i.e., overwrite) operations. Page write operations in a flash memory must be preceded by an erase operation and within a block pages need to be written sequentially. Read and write operations are performed in page-level, while erase operations are performed in block-level. In addition, before the erase is being done on a block, the valid (i.e., not over-written) pages from that block need to be moved to a new pre-erased blocks. Thus, a page update operation incurs lot of page read and write operations. The typical access latencies for read, write, and erase operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [9]. Besides the in-place update problem, flash memory exhibits another limitation – a flash block can only be erased for limited number of times (e.g., 10K-100K) [9].

The Flash Translation layer (FTL) is an intermediate software layer inside an SSD, which hides the limitations of flash memory and provides a disk like interface. FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. To emulate disk like in-place update operation for a logical page (L_p), the FTL writes data into a new physical page (P_p), maintains a mapping between logical pages and physical pages, and marks the previous physical location of L_p as invalid for future garbage collection. FTL uses various wear leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity [20]. Thus, FTL allows current disk based application to use SSD without any modifications. However, it needs to internally deal with current limitations of flash memory (i.e., constraint of erasing a block before overwriting a page in that block). Recent studies show that current FTL schemes are very effective for the workloads with sequential access write patterns. However, for the workloads with random access patterns, these schemes show very poor performance [21, 23, 26, 28, 32]. One of the design goals of ChunkStash is to use flash memory in an

FTL friendly manner.

3 Flash-assisted Inline Deduplication System

We follow the overall framework of production storage deduplication systems currently in the industry [42, 30]. Data chunks coming into the system are identified by their SHA-1 hash [34] and looked up in an index of currently existing chunks in the system (for that storage location or stream). If a match is found, the metadata for the file (or, object) containing that chunk is updated to point to the location of the existing chunk. If there is no match, the new chunk is stored in the system and the metadata for the associated file is updated to point to it. (In another variation, the chunk hash is included in the file/object metadata and is translated to chunk location during read access.) Comparing data chunks for duplication by their 20-byte SHA-1 hash instead of their full content is justified by the fact that the probability of SHA-1 hash match for non-identical chunks is less by many orders of magnitude than the probability of hardware error [37]. We allocate 44 bytes for the metadata portion. The 20-byte chunk hash is the key and the 44-byte metadata is the value, for a total key-value pair size of 64 bytes.

Similar to [42] and unlike [30], our system targets *complete deduplication* and ensures that no duplicate chunks exist in the system after deduplication. However, we also provide a technique for RAM usage reduction in our system that comes at the expense of marginal loss in deduplication quality.

We summarize the main components of the system below and then delve into the details of the chunk metadata store on flash which is a new contribution of this paper.

Data chunking: We use *Rabin fingerprinting* based sliding window hash [38] on the data stream to identify chunk boundaries in a content dependent manner. A chunk boundary is declared when the lower order bits of the Rabin fingerprint match a certain pattern. The length of the pattern can be adjusted to vary the average chunk size. The average chunk size in our system is 8KB as in [42]. *Ziv-Lempel compression* [43] on individual chunks can achieve an average compression ratio of 2:1, as reported in [42] and also verified on our datasets, so that the size of the stored chunks on hard disk averages around 4KB. The SHA-1 hash of a chunk serves as its chunk-id in the system.

On-disk Container Store: The *container store* on hard disk manages the storage of chunks. Each container stores at most 1024 chunks and averages in size around 4MB. (Because of the law of averages for this large number (1024) of chunks, the deviation of container size from this average is relatively small.) As new (unique) chunks come into the system, they are appended to the current container buffered in RAM. When the current container reaches the target size of 1024 chunks, it is sealed and written to hard disk and a new (empty) container is opened for future use.

Chunk Metadata Store on Flash (ChunkStash): To eliminate hard disk accesses for chunk-id lookups, we maintain, in flash, the metadata for all chunks in the system and index them using a specialized RAM index. The chunk metadata store on flash is a new contribution of this paper and is discussed in Section 3.1.

Chunk and Container Metadata Caches in RAM: A *cache for chunk metadata* is also maintained in RAM as in [42]. The fetch (prefetch) and eviction policies for this cache are executed at the container level (i.e., metadata for all chunks in a container). To implement this container level prefetch and eviction policy, we maintain a fixed size *container metadata cache* for the containers whose chunk metadata are currently held in RAM – this cache maps a container-id to the chunk-ids it contains. The size of the chunk metadata cache is determined by the size of the container metadata cache, i.e., for a container metadata cache size of C containers, the chunk metadata cache needs to hold $1024{*}b$ chunks. A distinguishing feature of ChunkStash (compared to the system in [42] is that it does not need to use bloom filters to avoid secondary storage (hard disk or flash) lookups for non-existent chunks.

Prefetching Strategy: We use the basic idea of predictability of sequential chunk-id lookups during second and subsequent full backups exploited in [42]. Since datasets do not change much across consecutive backups, duplicate chunks in the current full backup are very likely to appear in the same order as they did in the previous backup. Hence, when the metadata for a chunk is fetched from flash (upon a miss in the chunk metadata cache in RAM), we prefetch the metadata for all chunks in that container into the chunk metadata cache in RAM and add the associated container's entry to the container metadata cache in RAM. Because of this prefetching strategy, it is quite likely that the next several hundreds or thousand of chunk lookups will hit in the RAM chunk metadata cache.

RAM Chunk Metadata Cache Eviction Strategy: The container metadata cache in RAM follows a Least Recently Used (LRU) replacement policy. When a container is evicted from this cache, the chunk-ids of all the chunks it contains are removed from the chunk metadata cache in RAM.

3.1 ChunkStash: Chunk Metadata Store on Flash

As a new contribution of this paper, we present the architecture of ChunkStash, the on-flash chunk metadata store, and the rationale behind some design choices. The design of ChunkStash is driven by the need to work around two types of operations that are not efficient on flash media, namely:

1. **Random Writes:** Small random writes effectively need to update data portions within pages. Since a (physical) flash page cannot be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data on the page needs to be relocated to the new page.

2. **Writes less than flash page size:** Since a page is the smallest unit of write on flash, writing an amount less than a page renders the rest of the (physical) page wasted – any subsequent append to that partially written (logical) page will need copying of existing data and writing to a new (physical) page.

Given the above, the most efficient way to write flash is to simply use it as an append log, where an append operation involves one or more flash pages worth of data (current flash page size is typically 2KB or 4KB). This is the main constraint that drives the rest of our key-value store design. Flash has been used in a log-structured manner and its benefits reported in earlier work ([22, 41, 33, 15, 11]. We organize chunk metadata storage on flash into logical page units of 64KB which corresponds to the metadata for all chunks in a single container. (At 1024 chunks per container and 64 bytes per chunk-id and metadata, a container's worth of chunk metadata is 64KB in size.)

ChunkStash has the following main components, as shown in Figure 1:

RAM Chunk Metadata Write Buffer: This is a fixed-size data structure maintained in RAM that buffers chunk metadata information for the currently open container. The buffer is written to flash when the current container is sealed, i.e., the buffer accumulates 1024 chunk entries and reaches a size of 64KB. The RAM write buffer is sized to 2-3 times the flash page size so that chunk metadata writes can still go through when part of the buffer is being written to flash.

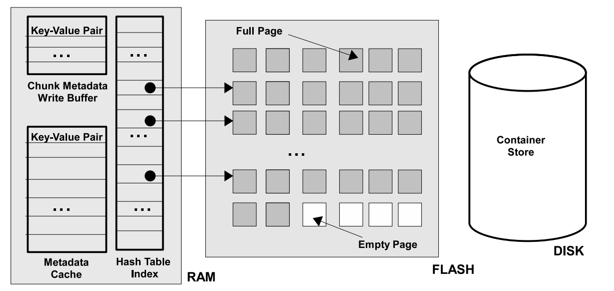


Figure 1: ChunkStash architectural overview.

RAM Hash Table (HT) Index: The index structure, for chunk metadata entries stored on flash, is maintained in RAM and is organized as a hash table with the design goal of one flash read per lookup. The index maintains pointers to the full (chunk-id, metadata) pairs stored on flash. Key features include resolving collisions using a variant of cuckoo hashing and storing compact key signatures in memory to tradeoff between RAM usage and false flash reads. We explain these aspects shortly.

On-Flash Store: The flash store provides persistent storage for chunk metadata and is organized as an append log. Chunk metadata is written (appended) to flash in units of a *logical page size* of 64KB, corresponding to the chunk metadata of a single container.

3.2 Hash Table Design for ChunkStash

We outline the salient aspects of the hash table design for ChunkStash.

Resolving hash collisions using cuckoo hashing: Hash function collisions on keys result in multiple keys mapping to the same hash table index slot – these need to be handled in any hashing scheme. Two common techniques for handling such collisions include *linear probing* and *chaining* [25]. Linear probing can increase lookup time arbitrarily due to long sequences of colliding slots. Chaining hash table entries in RAM, on the other hand, leads to increased memory usage, while chaining buckets of key-value pairs is not efficient for use with flash, since partially filled buckets will map to partially filled flash pages that need to be appended over time, which is not an efficient flash operation. Moreover, the latter will result in multiple flash page reads during key lookups and writes, which will reduce throughput.

ChunkStash structures the HT index as an array of slots and uses a variant of *cuckoo hashing* [35] to resolve collisions. Cuckoo hashing provides flexibility for each key to be in one of $n \geq 2$ candidate positions and for later inserted keys to relocate earlier inserted keys to any of their other candidate positions – this keeps the linear probing chain sequence upper bounded at n. In fact, the value proposition of cuckoo hashing is in increasing hash table load factors while keeping lookup times bounded to a constant. A study [44] has shown that cuckoo hashing is much faster than chained hashing as hash table load factors increase. The name "cuckoo" is derived from the behavior of some species of the cuckoo bird – the cuckoo chick pushes other eggs or young out of the nest when it hatches, much like the hashing scheme kicks previously inserted items out of their location as needed.

In the variant of cuckoo hashing we use, we work with n random hash functions $h_1, h_2, \ldots, h_n$ that are used to obtain n *candidate positions* for a given key x. These candidate position indices for key x are obtained from the lower-order bit values of $h_1(x), h_2(x), \ldots, h_n(x)$ corresponding to a modulo operation. During insertion, the key is inserted in the first available candidate slot. When all slots for a given key x are occupied during insertion (say, by keys $y_1, y_2, \ldots, y_n$), room can be made for key x by relocating keys y_i in these occupied slots, since each key y_i has a choice of $(n-1)$ other locations to go to.

In the original cuckoo hashing scheme [35], a recursive strategy is used to relocate one of the keys y_i – in the worst case, this strategy could take many key relocations or get into an infinite loop, the probability for which can be shown to be very small and decreasing exponentially in n [35]. In our design, the system attempts a small number of key relocations after which it makes room by picking a key to move to an auxiliary linked list (or, hash table). In practice, by dimensioning the HT index for a certain load factor and by choosing a suitable value of n, such events can be made extremely rare, as we investigate in Section 4.4. Hence, the size of this auxiliary data structure is small. The viability of this approach has also been verified in [24], where the authors show, through analysis and simulations, that a very small constant-sized auxiliary space can dramatically reduce the insertion failure probabilities associated with cuckoo hashing. (That said, we also want to add that the design of ChunkStash is amenable to other methods of hash table collision resolution.)

The number of hash function computations during lookups can be reduced from the worst case value of n to 2 using the standard technique of *double hashing* from the hashing literature [25]. The basic idea is that two hash functions g_1 and g_2 can simulate more than two hash functions of the form $h_i(x) = g_1(x) + ig_2(x)$. In our case, i will range from 0 to $n-1$. Hence, the use of higher number of hash functions in cuckoo hashing does not incur additional hash function computation overheads but helps to achieve higher hash table load factors.

Reducing RAM usage per slot by storing compact key signatures: Traditional hash table designs store the respective key in each entry of the hash table index [25]. Depending on the application, the key size could range from few tens of bytes (e.g., 20-byte SHA-1 hash as in storage deduplication) to hundreds of bytes or more. Given that RAM size is limited (commonly in the order of few to several gigabytes in servers) and is more expensive than flash (per GB), if we store the full key in each entry of the RAM HT index, it may well become the bottleneck for the maximum number of entries on flash that can be indexed from RAM before flash storage capacity bounds kick in. On the other hand, if we do not store the key at all in the HT index, the search operation on the HT index would have to follow HT index pointers to flash to determine whether the key stored in that slot matches the search key – this would lead to many *false flash reads*, which are expensive, since flash access speeds are 2-3 orders of magnitude slower than that of RAM.

To address the goals of maximizing HT index capacity (number of entries) and minimizing false flash reads, we store a *compact key signature* (order of few bytes) in each entry of the HT index. This signature is derived from *both the key and the candidate position number that it is stored at*. In ChunkStash, when a key x is stored in its candidate position number i, the signature in the respective HT index slot is derived from the higher order bits of the hash value $h_i(x)$. During a search operation, when a key y is looked up in its candidate slot number j, the respective signature is computed from $h_j(y)$ and compared with the signature stored in that slot. Only if a match happens is the pointer to flash followed to check whether the full key matches. We investigate the percentage of false reads as a function of the compact signature size in Section 4.4.

Storing key-value pairs on flash: Chunk-id and metadata pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. The HT index contains pointers to (chunk-id, metadata) pairs stored on flash. We use a 4-byte pointer, which is a combination of a logical page pointer and a page offset. With 64-byte key-value pair sizes, this is sufficient to index 256GB of chunk metadata on flash – for an average chunk size of 8KB, this corresponds to a maximum (deduplicated) storage dataset size of about 33TB. (ChunkStash reserves the all-zero pointer to indicate an empty HT index slot.)

3.3 Putting It All Together

To understand the hierarchical relationship of the different storage areas in ChunkStash, it is helpful to understand the sequence of accesses during inline deduplica-

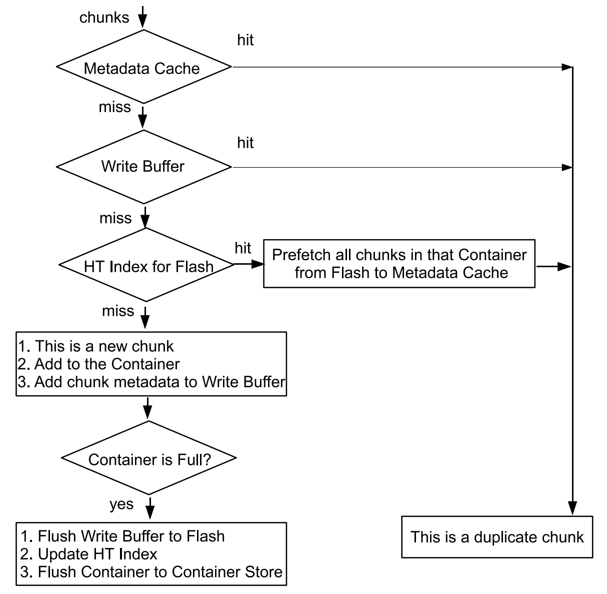

chunks
Metadata Cache
hit
miss
Write Buffer
hit
miss
HT Index for Flash
hit
Prefetch all chunks in that Container from Flash to Metadata Cache
miss
1. This is a new chunk
2. Add to the Container
3. Add chunk metadata to Write Buffer
Container is Full?
yes
1. Flush Write Buffer to Flash
2. Update HT Index
3. Flush Container to Container Store
This is a duplicate chunk

Figure 2: Flowchart of deduplication process in ChunkStash.

tion. A flowchart for this is provided in Figure 2. Recall that when a new chunk comes into the system, its SHA-1 hash is first looked up to determine if the chunk is a duplicate one. If not, the new chunk-id is inserted into the system.

A *chunk-id lookup* operation first looks up the RAM chunk metadata cache. Upon a miss there, it looks up the RAM chunk metadata write buffer. Upon a miss there, it searches the RAM HT Index in order to locate the chunk-id on flash. If the chunk-id is present on flash, its metadata, together with the metadata of all chunks in the respective container, is fetched into the RAM chunk metadata cache.

A *chunk-id insert* operation happens when the chunk coming into the system has not been seen earlier. This operation writes the chunk metadata into the RAM chunk metadata write buffer. The chunk itself is appended to the currently open container buffered in RAM. When the number of chunk entries in the RAM chunk metadata write buffer reaches the target of 1024 for the current container, the container is sealed and written to the container store on hard disk, and its chunk metadata entries are written to flash and inserted into the RAM HT index.

3.4 RAM and Flash Capacity Considerations

The indexing scheme in ChunkStash is designed to use a small number of bytes in RAM per key-value pair so as to maximize the amount of indexable storage on flash for a given RAM usage size. The RAM HT index capacity de-

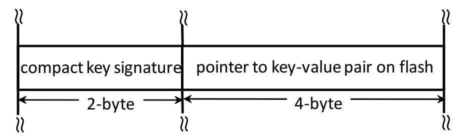


Figure 3: RAM HT Index entry and example sizes in ChunkStash. (The all-zero pointer is reserved to indicate an empty HT index slot.)

termines the number of chunk-ids stored on flash whose metadata can be accessed with one flash read. The RAM size for the HT index can be determined with application requirements in mind. With a 2-byte compact key signature and 4-byte flash pointer per entry, the RAM usage in ChunkStash is 6 bytes per entry as shown in Figure 3. For a given average chunk size, this determines the relationship among the following quantities – RAM and flash usage per storage dataset and associated storage dataset size.

For example, a typical RAM usage of 4GB per machine for the HT index accommodates a maximum of about 716 million chunk-id entries. At an average of 8KB size per data chunk, this corresponds to about 6TB of deduplicated data, for which the chunk metadata occupies about 45GB on flash. This flash usage is well within the capacity range of SSDs shipping in the market today (from 64GB to 640GB). When there are multiple such SSDs attached to the same machine, additional RAM is needed to fully utilize their capacity for holding chunk metadata. Moreover, RAM usage by the HT index in ChunkStash can be further reduced using techniques discussed in Section 3.5.

To reap the full performance benefit of ChunkStash for speeding up inline deduplication, it is necessary for the entire chunk metadata for the (current) backup dataset to fit in flash. Otherwise, when space on flash runs out, the append log will need to be *recycled* and written from the beginning. When a page on the flash log is rewritten, the earlier one will need to be evicted and the metadata contained therein written out to a hard disk based index. Moreover, during the chunk-id lookup process, if the chunk is not found in flash, it will need to be looked up in the index on hard disk. Thus, both the chunk-id insert and lookup pathways would suffer from the same bottlenecks of disk index based systems that we sought to eliminate in the first place.

ChunkStash uses flash memory to store chunk metadata and index it from RAM. It provides flexibility for flash to serve, or not to serve, as a permanent abode for chunk metadata for a given storage location. This decision can be driven by cost considerations, for example, because of the large gap in cost between flash memory and hard disk. When flash is not the permanent abode for chunk metadata for a given storage location, the chunk metadata log on flash can be written to hard disk in one large sequential write (single disk I/O) at the end of the backup process. At the beginning of the next full backup for this storage location, the chunk metadata log can be loaded back into flash from hard disk in one large sequential read (single disk I/O) and the containing chunks can be indexed in RAM HT index. This mode of operation amortizes the storage cost of metadata on flash across many backup datasets.

3.5 Reducing ChunkStash RAM Usage

The largest portion of RAM usage in ChunkStash comes from the HT index. This usage can be reduced by indexing in RAM only a small fraction of the chunks in each container (instead of the whole container). Flash will continue to hold metadata for *all* chunks in all containers, not just the ones indexed in RAM; hence when a chunk in the incoming data stream matches an indexed chunk, metadata for *all* chunks in that container will be prefetched in RAM. We use an uniform chunk sampling strategy, i.e., we index every i-th chunk in every container which gives a sampling rate of $1/i$.

Because only a subset of chunks stored in the system are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may, in fact, have appeared earlier and are already stored in the system. This will lead to some loss in deduplication quality, and hence, some amount of duplicate data chunks will be stored in the system. In Section 4.6, we study the impact of this on deduplication quality (and backup throughput). We find that the loss in deduplication quality is marginal when about 1% of the chunks in each container are indexed and becomes negligibly small when about 10% of the chunks are indexed. The corresponding RAM usage reductions for the HT index are appreciable at 99% and 90% respectively. Hence, indexing chunk subsets in ChunkStash provides a powerful knob for reducing RAM usage with only marginal loss in deduplication quality.

In an earlier approach for reducing RAM usage requirements of inline deduplication systems, the method of sparse indexing [30] chops the incoming data into multiple megabyte segments, samples chunks *at random* within a segment (based on the most significant bits of the SHA-1 hash matching a pattern, e.g., all 0s), and uses these samples to find few segments seen in the recent past that share many chunks. In contrast, our sampling method is deterministic and samples chunks at uniform intervals in each container for indexing. Moreover, we are able to match incoming chunks with sampled chunks in *all* containers stored in the system, not just those seen in the recent past. In our evaluations in Section 4.6, we show that our uniform sampling strategy gives better deduplication quality than random sampling (for the same sampling rate).

4 Evaluation

We evaluate the backup throughput performance of a ChunkStash based inline deduplication system and compare it with our implementation of a disk index based system as in [42]. We use three enterprise datasets and two full backups for our evaluations.

4.1 C# Implementation

We have prototyped ChunkStash in approximately 8000 lines of C# code. MurmurHash [5] is used to realize the the hash functions used in our variant of cuckoo hashing to compute hash table indices and compact signatures for keys; two different seeds are used to generate two different hash functions in this family for use with the double hashing based simulation of n hash functions, as described in Section 3.2. In our implementation, writes to the on-disk container store are performed in a non-blocking manner using a small pool of file writer worker threads. The metadata store on flash is maintained as a log file in the file system and is created/opened in non-buffered mode so that *there are no buffering/caching/prefetching effects in RAM from within the operating system.*

4.2 Comparison with Hard Disk Index based Inline Deduplication

We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [42]. The index used in [42] appears to be proprietary and no details are provided in the paper. Hence, for purposes of comparative evaluation, we have built a hard disk index based system incorporating the ideas in [42] with the hard disk index implemented by BerkeleyDB [1], an embedded key-value database that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include a "hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms.

BerkeleyDB does not use flash aware algorithms but we used the parameter settings recommended in [3] to improve its performance with flash. We use BerkeleyDB in its non-transactional concurrent data store mode that supports a single writer and multiple readers [40]. This mode does *not* support a transactional data store with the ACID properties, hence provides a fair comparison with ChunkStash . BerkeleyDB provides a choice of B-tree and hash table data structures for building indexes – we use the hash table version which we found to run faster. We use the C++ implementation of BerkeleyDB with C# API wrappers [2].

Trace	Size (GB)	Total Chunks	#Full Backups
Dataset 1	8GB	1.1 million	2
Dataset 2	32GB	4.1 million	2
Dataset 3	126GB	15.4 million	2

Table 1: Properties of the three traces used in the performance evaluation of ChunkStash. The average chunk size is 8KB.

4.3 Evaluation Platform and Datasets

We use a standard server configuration to evaluate the inline deduplication performance of ChunkStash and compare it with the disk index based system that uses BerkeleyDB. The server runs Windows Server 2008 R2 and uses an Intel Core 2 Duo E6850 3GHz CPU, 4GB RAM, and fusionIO 160GB flash drive [4] attached over PCIe interface. Containers are written to a RAID4 system using five 500GB 7200rpm hard disks. A separate hard disk is used for storing disk based indexes. For the fusionIO drive, *write buffering inside the device is disabled* and cannot be turned on through operating system settings. The hard drives used support write buffering inside the device by default and this setting was left on. This clearly gives some advantage to the hard disks for the evaluations but makes our comparisons of flash against hard disk more conservative.

To obtain traces from backup datasets, we have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk size, and compute the number of deduplicated chunks and storage bytes. The enterprise data backup traces we use for evaluations in this paper were obtained by our storage deduplication analysis tool using 8KB (average) chunk sizes (this is also the chunk size used in [30]). We obtained two full backups for three different storage locations, indicated as Datasets 1, 2, and 3 in Table 1. The number of chunks in each dataset (for each full backup) are about 1 million, 4 million, and 15 million respectively.

We compare the throughput (MB/sec processed from the input data stream) on the three traces described in Table 1 for the following four inline deduplication systems:

- Disk based index (BerkeleyDB) and RAM bloom filter [42] (***Zhu08-BDB-HDD***),
- Zhu08-BDB system with SSD replacement for BerkeleyDB index storage (***Zhu08-BDB-SSD***),
- Flash based index using ChunkStash (***ChunkStash-SSD***),
- ChunkStash with the SSD replaced by hard disk (***ChunkStash-HDD***).

Some of the design decisions in ChunkStash also work well when the underlying storage is hard disk and not flash (e.g., log structured data organization and sequential writes). Hence, we have included Chunkstash running on hard disk as a comparison point so as to bring out the impact of log structured organization for a store on hard disk for the storage deduplication application. (Note that BerkeleyDB does not use a log structured storage organization.)

All four systems use RAM prefetching techniques for the chunk index as described in [42]. When a chunk hash lookup misses in RAM (but hits the index on hard disk or flash), metadata for all chunks in the associated container are prefetched into RAM. The RAM chunk metadata cache size for holding chunk metadata is fixed at 20 containers, which corresponds to a total of 20,480 chunks. In order to implement the prefetching of container metadata in a single disk I/O for Zhu08-BDB, we maintain, in addition to the BerkeleyDB store, an auxiliary sequential log of chunk metadata that is appended with container metadata whenever a new container is written to disk.

Note that unlike in [42], our evaluation platform is not a production quality storage deduplication system but rather a research prototype. Hence, our throughput numbers should be interpreted in a relative sense among the four systems above, and not used for absolute comparison with other storage deduplication systems.

4.4 Tuning Hash Table Parameters

Before we make performance comparisons of ChunkStash with the disk index based system, we need to tune two parameters in our hash table design from Section 3.2, namely, (a) number of hash functions used in our variant of cuckoo hashing, and (b) size of compact key signature. These affect the throughput of read key and write key operations in different ways that we discuss shortly. For this set of experiments, we use one of the storage deduplication traces in a modified way so as to study the performance of hash table insert and lookup operations separately. We pre-process the trace to extract a set of 1 million unique keys, then insert all the key-value pairs, and then read all these key-value pairs in random order.

As has been explained earlier, the value proposition of cuckoo hashing is in accommodating higher hash table load factors without increasing lookup time. It has been

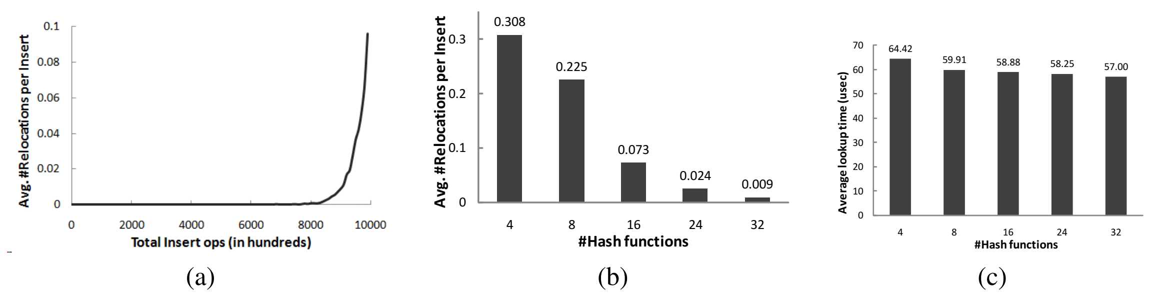


Figure 4: Tuning hash table parameters in ChunkStash: (a) Average number of relocations per insert as keys are inserted into hash table; (b) Average number of relocations per insert, and (c) Average lookup time (μsec), vs. number of hash functions (n), averaged between 75%-90% load factors.

shown mathematically that with 3 or more hash functions and with load factors up to 91%, insertion operations succeed in expected constant time [35]. With this prior evidence in hand, we target a maximum load factor of 90% for our cuckoo hashing implementation. Hence, for the dedup trace used in this section with 1 million keys, we fix the number of hash table slots to 1.1 million.

Number of Hash Functions. When the number of hash functions n is small, the performance of insert operations can be expected to degrade in two ways as the hash table loads up. First, an insert operation will find all its n slots occupied by other keys and the number of cascaded key relocations required to complete this insertion will be high. Since each key relocation involves a flash read (to read the full key from flash and compute its candidate positions), the insert operation will take more time to complete. Second, with an upper bound on the number of allowed key relocations (which we set to 100 for this set of experiments), the insert operation could lead to a key being moved to the auxiliary linked list – this increases the RAM space usage of the linked list as well as its average search time. On the other hand, as the number of hash functions increase, lookup times will increase because of increasing number of hash table positions searched. However, the latter undesirable effect is not expected to be as degrading as those for inserts, since a lookup in memory takes orders of magnitude less time than a flash read. We study these effects to determine a suitable number of hash functions for our design. (Note that because of our use of double hashing, the number of hash function computations per lookup does not increase with n.)

In Figure 4(a), we plot the average number of key relocations (hence, flash reads) per insert operation as keys are inserted into the hash table (for $n = 24$ hash functions). We see that the performance of insert operations degrades as the hash table loads up, as expected because of the impact of the above effect. Accordingly, for the following plots in this section, we present average numbers between 75% and 90% load factors as the insert operations are performed.

In Figure 4(b), we plot the average number of key relocations per insert operation as the number of hash functions n is varied, $n = 4, 8, 16, 24, 32$. Beyond $n = 16$ hash functions, the hash table incurs less than 0.1 key relocations (hence, flash reads) per insert operation. Figure 4(c) shows that there is no appreciable increase in average lookup time as the number of hash functions increase. (The slight decrease in average lookup time with increasing number of hash functions can be attributed to faster search times in the auxiliary linked list, whose size decreases as number of hash functions increases.)

Based on these effects, we choose $n = 24$ hash functions in the RAM HT Index for our ChunkStash implementation. Note that during a key lookup, all n hash values on the key need not be computed, since the lookup stops at the candidate position number the key is found in. Moreover, because of the use of double hashing, at most two hash function computations are incurred even when all candidate positions are searched for a key. We want to add that using fewer hash functions may be acceptable depending on overall performance requirements, but we do not recommend a number below $n = 8$. Also, with $n = 24$ hash functions, we observed that it is sufficient to set the maximum number of allowed key relocations (during inserts) to 5-10 to keep the number of inserts that go to the linked list very small.

Compact Key Signature Size. As explained in Section 3.1, we store compact key signatures in the HT index (instead of full keys) to reduce RAM usage. However, shorter signatures lead to more *false flash reads* during lookups (i.e., when a pointer into flash is followed because the signature in HT index slot matches, but the full key on flash does not match with the key being looked up). Since flash reads are expensive compared to RAM reads, the design should strike a balance between reducing false flash reads and RAM usage. We observe that the fraction of false reads is about 0.01% when the number of signature bytes is fixed at 2, and we use this signature

Trace	RAM Hit Rate	
	1st Full Backup	2nd Full Backup
Dataset 1	20%	97%
Dataset 2	2%	88%
Dataset 3	23%	80%

Table 2: RAM hit rates for chunk hash lookups on the three traces for each full backup.

size in our implementation. Even a 1-byte signature size may be acceptable given that only 0.6% of the flash reads are false in that case.

4.5 Backup Throughput

We ran the chunk traces from the the three datasets outlined in Table 1 on ChunkStash and our implementation of the system in [42] using BerkeleyDB as the chunk index on either hard disk or flash. We log the backup throughput (MB of data backed up per second) at a period of every 10,000 input chunks during each run and then take the overall average over a run to obtain throughput numbers shown in Figures 5, 6, and 7.

ChunkStash achieves average throughputs of about 190 MB/sec to 265 MB/sec on the first full backups of the three datasets. The throughputs are about 60%-140% more for the second full backup compared to the first full backup for the datasets – this reflects the effect of prefetching chunk metadata to exploit sequential predictability of chunk lookups during second full backup. The speedup of ChunkStash over Zhu08-BDB-HDD is about 30x-60x for the first full backup and 7x-40x for the second full backup. Compared to the Zhu08-BDB-SSD in which the hard disk is replaced by SSD for index storage, the speedup of ChunkStash is about 3x-4x for the first full backup and about 2x-4x for the second full backup. The latter reflects the relative speedup of ChunkStash due to the use of flash-aware data structures and algorithms over BerkeleyDB which is not optimized for flash device properties.

We also run ChunkStash on hard disk (instead of flash) to bring out the performance impact of a log-structured organization on hard disk. Sequential writes of container metadata to the end of the metadata log is a good design for and benefits hard disks also. On the other hand, lookups in the log from the in-memory index may involve random reads in the log which are expensive on hard disks due to seeks – however, container metadata prefetching helps to reduce the number of such random reads to the log. We observe that the throughput of ChunkStash-SSD is more than that of ChunkStash-HDD by about 50%-80% for the first full backup and by about 10%-20% for the second full backup for the three datasets.

In Table 2, we show the RAM hit rates for chunk hash

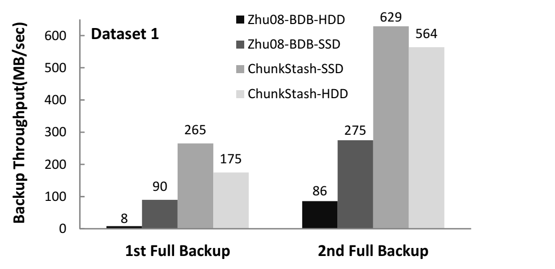


Figure 5: Dataset 1: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

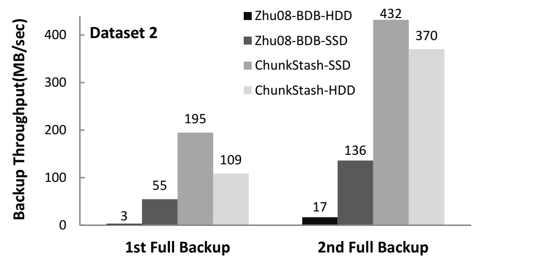


Figure 6: Dataset 2: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

lookups on the three traces for each full backup. The RAM hit rate for the first full backup is indicative of the redundancy (duplicate chunks) *within* the dataset – this is higher for Datasets 1 and 3 and is responsible for their higher backup throughputs. The RAM hit rate for the second full backup is indicative of its similarity with the first full backup – this manifests itself during the second full backup through fewer containers written and through sequential predictability of chunk hash lookups (which is exploited by the RAM prefetching strategy).

When compared to ChunkStash, the relatively worse performance of a BerkeleyDB based chunk metadata index can be attributed to the increased number of disk I/Os (random reads and writes). We measured the number of disk I/Os for Zhu08-BDB and ChunkStash systems using Windows Performance Analysis Tools (`xperf`) [8]. In Figure 8, we observe that the number of read I/Os in BerkeleyDB is 3x-7x that of ChunkStash and the number of write I/Os is about 600-1000x that of ChunkStash. Moreover, these writes I/Os in BerkeleyDB are all random I/Os, while ChunkStash is designed to use only sequential writes (appends) to the chunk metadata log. Because there is no locality in the key space in an application like storage deduplication, container metadata writes to a BerkeleyDB based index lead to in-place updates (random writes) to many different pages (on disk or flash) and appears to be one of the main reasons for its worse performance.

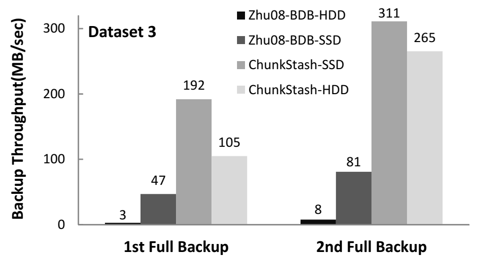


Figure 7: Dataset 3: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

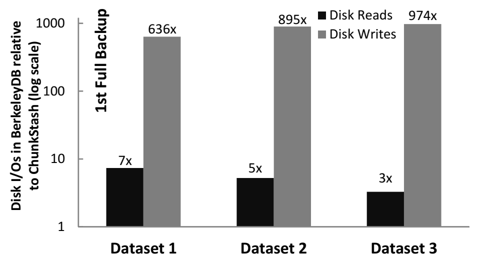


Figure 8: Disk I/Os (reads and writes) in BerkeleyDB relative to ChunkStash on the first full backup of the three datasets. (Note that the y-axis is log scale.)

4.6 Impact of Indexing Chunk Subsets

In Section 3.5, we saw that RAM usage in ChunkStash can be reduced by indexing a small fraction of the chunks in each container. In this section, we study the impact of this on deduplication quality and backup throughput. Because only a subset of the chunks are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may have appeared earlier and are already stored in the system. Hence, some amount of duplicate data chunks will be stored in the system. We compare two chunk sampling strategies – uniform chunk sampling strategy (i.e., index every i-th chunk) and random sampling (based on the most significant bits of the chunk SHA-1 hash matching a pattern, e.g., all 0s), the latter being used as part of a sparse indexing scheme in [30].

In Figure 9, we plot the fraction of chunks that are declared as new by the system during the second full backup of Dataset 2 as a percentage of the total number of chunks in the second full backup. The lower the value of this fraction, the better is the deduplication quality (the baseline for comparison being the case when all chunks are indexed). The fraction of chunks indexed in each container is varied as $1.563\% = 1/64$, $6.25\% = 1/16$, $12.5\% = 1/8$, and 100% (when all chunks are indexed). We choose sampling rates that are reciprocals of powers of 2 because those are the types of sampling rates

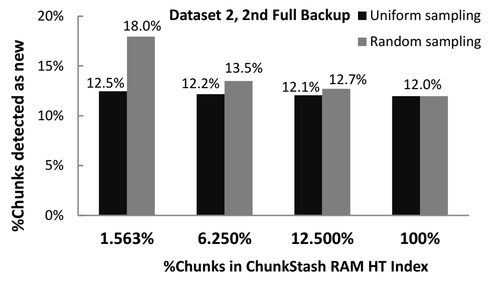


Figure 9: Dataset 2: Number of chunks detected as new as a fraction of the total number of chunks (indicating deduplication quality) vs. fraction of chunks indexed in ChunkStash RAM HT index in second full backup. (When 100% of the chunks are indexed, all duplicate chunks are detected accurately.) The x-axis fractions correspond to sampling rates of $1/64$, $1/16$, and $1/8$. For a sampling rate of $1/2^n$, uniform sampling indexes every 2^n-th chunk in a container, whereas random sampling indexes chunks with first n bits of SHA-1 hash are all 0s.

possible in the random sampling scheme – when n most significant bits of the SHA-1 hash are matched to be all 0s, the sampling rate is $1/2^n$.

We observe that when 1.563% of the chunks are indexed, the uniform chunk sampling strategy results in only a 0.5% increase in the number of chunks detected as new (as a fraction of the whole dataset). This loss in deduplication quality could be viewed as an acceptable tradeoff in exchange for a 98.437% reduction in RAM usage of ChunkStash HT index. When 12.5% of the chunks are indexed, the loss in deduplication quality is almost negligible at 0.1% (as a fraction of the whole dataset), but the reduction in RAM usage of ChunkStash HT index is still substantial at 87.5%. Hence, indexing chunk subsets provides a powerful knob for reducing RAM usage in ChunkStash with only marginal loss in deduplication quality.

We also note that the loss in deduplication quality with random sampling is worse than that with uniform sampling, especially at lower sampling rates. An intuitive explanation for this is that uniform sampling gives better coverage of the input stream at regular intervals of number of chunks (hence, data size intervals), whereas random sampling (based on some bits in the chunk SHA-1 hash matching a pattern) could lead to large gaps between two successive chunk samples in the input stream.

An interesting side-effect of indexing chunk subsets is the increase in backup throughput – this is shown in Figure 10 for the first and second full backups for Dataset 2. The effects on first and second full backups can be explained separately. When a fraction of the chunks are indexed, chunk hash keys are inserted into the RAM HT index at a lower rate during the first full backup, hence the throughput increases. (Note, however, that writes to

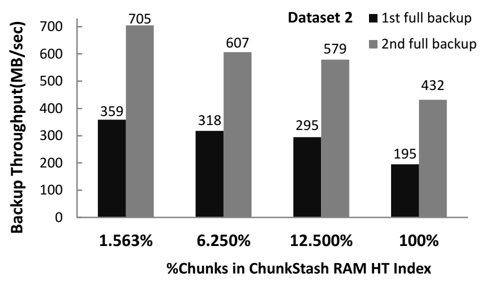


Figure 10: Dataset 2: Backup throughput (MB/sec) vs. fraction of chunks indexed in ChunkStash RAM HT index in first and second full backups.

the metadata log on flash are still about the same, but possibly slightly higher due to less accurate detection of duplicate chunks within the dataset.) During the second full backup, fewer chunks from the incoming stream are found in the ChunkStash RAM HT index, hence the number of flash reads during the backup process are reduced, leading to higher throughput. Both of these benefits drop gradually as more chunks are indexed but still remain substantial at a sampling rate of 12.5% – with the loss in deduplication quality being negligible at this point, the tradeoff is more of a win-win situation than a compromise involving the three parameters of RAM usage (low), deduplication throughput (high), and loss is deduplication quality (negligible).

4.7 Flash Memory Cost Considerations

Because flash memory is more expensive per GB than hard disk, we undertake a performance/dollar analysis in an effort to mitigate cost concerns about a flash-assisted inline deduplication system like ChunkStash. In our system, we use 8KB (average) chunk sizes and store them compressed on hard disk – with an average compression ratio of 2:1 (which we verified for our datasets), the space occupied by a data chunk on hard disk is about 4KB. With chunk metadata size of 64 bytes, the metadata portion is about $64/(4 * 1024) = 1/64$ fraction of the space occupied by chunk data on hard disk. With flash being currently 10 times more expensive per GB than hard disk, the cost of metadata storage on flash is about $10/64 = 16\%$ that of data storage on HDD. Hence, the overall increase in storage cost is about 1.16x.

Using a ballpark improvement of 25x in backup throughput for ChunkStash over disk based indexes for inline deduplication (taken from our evaluations reported in Section 4.5), *the gain in performance/dollar for ChunkStash over disk based indexes is about* $25/1.16 = 22x$. We believe that this justifies the additional capital investment in flash for inline deduplication systems that are hard-pressed to meet short backup window deadlines.

Moreover, the metadata storage cost on flash can be amortized across many backup datasets by storing a dataset's chunk metadata on hard disk and loading to flash just before the start of the backup process for the respective dataset – this reduces the flash memory investment in the system and makes the performance-cost economics even more compelling.

5 Related Work

We review related work that falls into two categories, namely, storage deduplication and key-value store on flash. The use of flash memory to speed up inline deduplication is a unique contribution of our work – their is no prior research that overlaps both of these areas. We also make new contributions in the design of ChunkStash, the chunk metadata store on flash which can be used as a key-value store for other applications as well.

5.1 Storage Deduplication

Zhu et al.'s work [42] is among the earliest research in the inline storage deduplication area and provides a nice description of the innovations in Data Domain's production storage deduplication system. They present two techniques that aim to reduce lookups on the disk-based chunk index. First, a bloom filter [13] is used to track the chunks seem by the system so that disk lookups are not made for non-existing chunks. Second, upon a chunk lookup miss in RAM, portions of the disk-based chunk index (corresponding to all chunks in the associated container) are prefetched to RAM. The first technique is effective for new data (e.g., first full backup) while the second technique is effective for little or moderately changed data (e.g., subsequent full backups). Their system provides perfect deduplication quality. Our work aims to reduce the penalty of index lookup misses in RAM that go to hard disk by orders of magnitude by designing a flash-based index for storing chunk metadata.

Lillibridge et al. [30] use the technique of sparse indexing to reduce the in-memory index size for chunks in the system at the cost of sacrificing deduplication quality. The system chunks the data into multiple megabyte segments, which are then lightly sampled (at random based on the chunk SHA-1 hash matching a pattern), and the samples are used to find a few segments seen in the recent past that share many chunks. Obtaining good deduplication quality depends on the chunk locality property of the dataset – whether duplicate chunks tend to appear again together with the same chunks. When little or no chunk locality is present, the authors recommend an approach based on *file similarity* [12] that achieves significantly better deduplication quality. In our work, the memory usage of ChunkStash can be reduced by indexing only a subset of the chunk metadata on flash (using an uniform sampling strategy, which we found gives better deduplication quality than random sampling).

DEDE [16] is a decentralized deduplication system designed for SAN clustered file systems that supports a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim storage for duplicate blocks. Unlike inline deduplication systems, the deduplication process is done out-of-band so as to minimize its impact on file system performance. In this paper, we focus on inline storage deduplication systems.

HYDRAstor [17] discusses architecture and implementation of a commercial secondary storage system, which is content addressable and implements a global data deduplication policy. Recently, a new file system, called HydraFS [39], has been designed for HYDRAstor. In order to reduce the disk accesses, HYDRAstor uses bloom filter [13] in RAM. In contrast, we aim to eliminate disk seek/access (and miss) overheads by using a flash-based chunk metadata store.

Deduplication systems differ in the granularity at which they detect duplicate data. EMC's Centera [18] uses file level duplication, LBFS [31] uses variable-sized data chunks obtained using Rabin fingerprinting, and Venti [36] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [19] and bimodal chunking algorithm [27] produce variable-sized chunks.

5.2 Key-Value Store on Flash

Flash memory has received lots of recent interest as a stable storage media that can overcome the access bottlenecks of hard disks. Researchers have considered modifying existing applications to improve performance on flash as well as providing operating system support for inserting flash as another layer in the storage hierarchy. In this section, we briefly review work that is related to key-value store aspect of ChunkStash and point out its differentiating aspects.

MicroHash [41] designs a memory-constrained index structure for flash-based sensor devices with the goal of optimizing energy usage and minimizing memory footprint. This work does not target low latency operations as a design goal – in fact, a lookup operation may need to follow chains of index pages on flash to locate a key, hence involving multiple flash reads.

FlashDB [33] is a self-tuning B^+-tree based index that dynamically adapts to the mix of reads and writes in the workload. Like MicroHash, this design also targets memory and energy constrained sensor network devices. Because a B^+-tree needs to maintain partially filled leaf-level buckets on flash, appending of new keys to these buckets involves random writes, which is not an efficient flash operation. Hence, an adaptive mechanism is also provided to switch between disk and log-based modes. The system leverages the fact that key values in sensor applications have a small range and that at any given time, a small number of these leaf-level buckets are active. Minimizing latency is not an explicit design goal.

The benefits of using flash in a log-like manner have been exploited in FlashLogging [15] for synchronous logging. This system uses multiple inexpensive USB drives and achieves performance comparable to flash SSDs but with much lower price. Flashlogging assumes sequential workloads.

Gordon [14] uses low power processors and flash memory to build fast power-efficient clusters for data-intensive applications. It uses a flash translation layer design tailored to data-intensive workloads. In contrast, ChunkStash builds a persistent key-value store using existing flash devices (and their FTLs) with throughput maximization as the main design goal.

FAWN [11] uses an array of embedded processors equipped with small amounts of flash to build a power-efficient cluster architecture for data-intensive computing. Like ChunkStash, FAWN also uses an in-memory hash table to index key-value pairs on flash. The differentiating aspects of ChunkStash include its adaptation for the specific server-class application of inline storage deduplication and in its use of a specialized in-memory hash table structure with cuckoo hashing to achieve high hash table load factors (while keeping lookup times bounded) and reduce RAM usage.

BufferHash [10] builds a content addressable memory (CAM) system using flash storage for networking applications like WAN optimizers. It buffers key-value pairs in RAM, organized as a hash table, and flushes the hash table to flash when the buffer is full. Past copies of hash tables on flash are searched using a time series of Bloom filters maintained in RAM and searching keys on a given copy involve multiple flash reads. Moreover, the storage of key-value pairs in hash tables on flash wastes space on flash, since hash table load factors need to be well below 100% to keep lookup times bounded. In contrast, ChunkStash is designed to access any key using one flash read, leveraging cuckoo hashing and compact key signatures to minimize RAM usage of a customized in-memory hash table index.

6 Conclusion

We designed ChunkStash to be used as a high throughput persistent key-value storage layer for chunk metadata for inline storage deduplication systems. To this end, we

incorporated flash aware data structures and algorithms into ChunkStash to get the maximum performance benefit from using SSDs. We used enterprise backup datasets to drive and evaluate the design of ChunkStash . Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 7x-60x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-4x. Building on the base design, we also show that the RAM usage of ChunkStash can be reduced by 90-99% with only a marginal loss in deduplication quality.

References

[1] BerkeleyDB. `http://www.oracle.com/technology/products/berkeley-db/index.html`.

[2] BerkeleyDB for .NET. `http://sourceforge.net/projects/libdb-dotnet/`.

[3] BerkeleyDB Memory-only or Flash configurations. `http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html`.

[4] Fusion-IO Drive Datasheet. `http://www.fusionio.com/PDFs/Data_Sheet_ioDrive_2.pdf`.

[5] MurmurHash Function. `http://en.wikipedia.org/wiki/MurmurHash`.

[6] MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers. `http://www.fusionio.com/case-studies/myspace-case-study.pdf`.

[7] Releasing Flashcache. `http://www.facebook.com/note.php?note_id=388112370932`.

[8] Windows Performance Analysis Tools (xperf). `http://msdn.microsoft.com/en-us/performance/cc825801.aspx`.

[9] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX*, 2008.

[10] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI*, 2010.

[11] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *SOSP*, Oct. 2009.

[12] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *MASCOTS*, 2009.

[13] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics*, 2002.

[14] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *ASPLOS*, 2009.

[15] S. Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *SIGMOD*, 2009.

[16] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX*, 2009.

[17] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, , and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *FAST*, 2009.

[18] EMC Corporation. EMC Centera: Content Addresses Storage System, Data Sheet, April 2002.

[19] K. Eshghi. A framework for analyzing and improving content-based chunking algorithms. *HP Labs Technical Report HPL-2005-30 (R.1)*, 2005.

[20] E. Gal and S. Toledo. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys*, volume 37, 2005.

[21] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. In *ASPLOS*, 2009.

[22] A. Kawaguchi, S. Nishioka, and H. Motoda. A Flash-Memory Based File System. In *USENIX*, 1995.

[23] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *FAST*, 2008.

[24] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

[25] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, Reading, MA, 1998.

[26] I. Koltsidas and S. Viglas. Flashing Up the Storage Layer. In *VLDB*, 2008.

[27] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *FAST*, 2010.

[28] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song. A Log Buffer-Based Flash Translation Layer Using Fully-Associate Sector Translation. In *ACM TECS*, volume 6, 2007.

[29] A. Leventhal. Flash Storage Memory. *Communications of the ACM*, 51(7):47 – 51, July 2008.

[30] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST*, 2009.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, 2001.

[32] S. Nath and P. Gibbons. Online Maintenance of Very Large Random Samples on Flash Storage. In *VLDB*, 2008.

[33] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN*, 2007.

[34] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[35] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[36] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *FAST*, 2002.

[37] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *FAST*, 2002.

[38] M. O. Rabin. Fingerprinting by Random Polynomials. *Harvard University Technical Report TR-15-81*, 1981.

[39] C. Ungureanu, B. Atkin, A. Aranya, S. R. Salil Gokhale, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST*, 2010.

[40] H. Yadava. *The Berkeley DB Book*. Apress, 2007.

[41] D. Zeinalipour-yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *FAST*, 2005.

[42] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST*, 2008.

[43] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337 – 343, May 1977.

[44] M. Zukowski, S. Heman, and P. Boncz. Architecture-Conscious Hashing. In *DAMON*, 2006.

Mining Invariants from Console Logs for System Problem Detection

Jian-Guang LOU[1] Qiang FU[1] Shengqi YANG[2] Ye XU[3] Jiang LI[1]

[1]Microsoft Research Asia
Beijing, P. R. China
{jlou, qifu, jiangli}@microsoft.com

[2]Dept. of Computer Science
Beijing Univ. of Posts and Telecom
v-sheyan@microsoft.com

[3]Dept. of Computer Science
Nanjing University, P.R. China
v-yexu@microsoft.com

Abstract

Detecting execution anomalies is very important to the maintenance and monitoring of large-scale distributed systems. People often use console logs that are produced by distributed systems for troubleshooting and problem diagnosis. However, manually inspecting console logs for the detection of anomalies is unfeasible due to the increasing scale and complexity of distributed systems. Therefore, there is great demand for automatic anomaly detection techniques based on log analysis. In this paper, we propose an unstructured log analysis technique for anomaly detection, with a novel algorithm to automatically discover program invariants in logs. At first, a log parser is used to convert the unstructured logs to structured logs. Then, the structured log messages are further grouped to log message groups according to the relationship among log parameters. After that, the program invariants are automatically mined from the log message groups. The mined invariants can reveal the inherent linear characteristics of program work flows. With these learned invariants, our technique can automatically detect anomalies in logs. Experiments on Hadoop show that the technique can effectively detect execution anomalies. Compared with the state of art, our approach can not only detect numerous real problems with high accuracy but also provide intuitive insight into the problems.

1 Introduction

Most software systems generate console log messages for troubleshooting. The console log messages are usually unstructured free-form text strings, which are used to record events or states of interest and to capture the system developers' intent. In general, when a job fails, experienced system operators examine recorded log files to gain insight about the failure, and to find the potential root causes. Especially for debugging distributed systems, checking the console logs to locate system problems is the most applicable way because the instrumentation or dump based approaches may make a system behave differently from its daily execution and introduce overhead.

We are now facing an explosive growth of large-scale Internet services that are supported by a set of large server clusters. The trend of cloud computing also drives the deployment of large-scale data centers. Typical systems such as those of Google, Amazon and Microsoft consist of thousands of distributed components including servers, network devices, distributed computing software components, and operating systems. Due to the increasing scale and complexity of these distributed systems, it becomes very time consuming for a human operator to diagnose system problems through manually examining a great amount of log messages. Therefore, automated tools for problem diagnosis through log analysis are essential for many distributed systems.

Several research efforts have been made in the design and development of automatic tools for log analysis. Most of the traditional automatic tools detect system problems by checking logs against a set of rules that describe normal system behaviors. Such rules are manually predefined by experts according to their knowledge about system design and implementation. SEC [1], Logsurfer [2] and Swatch [3] are three typical examples of a rule-based log analysis tool. However, it is very expensive to manually define such rules because a great amount of system experts' efforts are required. Besides, a modern system often consists of multiple components developed by different groups or even different companies, and a single expert may not have complete knowledge of the system; therefore, constructing the rules needs close cooperation of multiple experts, which brings more difficulties and costs. Furthermore, after each upgrade of the system, the experts need to check or modify the predefined rules again. In summary, manually defining rules for detecting problems from logs is expensive and inefficient.

Recently, there have appeared some statistic learning based automatic tools that analyze console logs, profiles and measurements for system monitoring and trouble shooting. Such approaches extract features from logs, traces or profiles, then use statistical techniques, such as subspace analysis [5, 6, 13], clustering and classification algorithms [7, 8], to automatically build models, and then identify failures or problems according to the learned models. However, most of the learned models are black box models that cannot be easily understood by human operators [5]. They may detect anomalies in a high dimensional feature space, but can hardly provide intuitive and meaningful explanations for the detected anomalies.

In this paper, we aim to automatically mine constant linear relationships from console logs based on a statistical learning technique. Such relationships that always hold in system logs under different inputs and workloads are considered as program invariants. These linear relationships can capture the normal program execution behavior. If a new log breaks certain invariants, we say an anomaly occurs during the system execution. Here is a simple example of invariant: in the normal executions of a system, the number of log messages indicating "Open file" is usually equal to the number of log messages corresponding to "Close file", because each opened file will be closed at some stage eventually. Such relationships are often well utilized when we manually check problems in log files. If it is broken, the operator can know there must be a system anomaly of the file operation (e.g. file handler leak) and safely speculate where the problem is. With this observation, we propose an approach to automatically detect system anomalies based on mining program invariants from logs. Unlike other statistical based approaches, program invariant has a very clear physical meaning that can be easily understood by human operators. It can not only detect system anomalies but also give a meaningful interpretation for each detected anomaly. Such interpretation associates the anomaly with the execution logic, which can significantly help system operators to diagnose system problems.

In our approach, we first convert unstructured log messages to structured information, including message signatures and parameters, by using a log parser. Then, the messages are grouped based on the log parameters. Based on the message groups, we discover sparse and integer invariants through a hypothesis and testing framework. In addition, the scalability and efficiency issues of the invariant search algorithm are discussed, and some techniques to reduce the computational cost are introduced. In brief, the main contribution of our work can be summarized as follows:

- We propose a method to automatically identify a set of parameters that correspond to the same program variable (namely cogenetic) based on the parameter value range analysis.
- We propose a method to discover sparse and integer invariants that have very clear physical meanings associated with system execution. The computational complexity of our algorithm can be significantly reduced to fit real-world large-scale applications.
- We apply the mined invariants to detect system anomalies. By checking the broken invariants, our method can provide helpful insights about execution problems.

The paper is organized as follows. In Section 2, we briefly introduce the previous work that is closely related to ours. Section 3 provides the basic idea and an overview of our approach. In Section 4, we briefly describe the log parsing method that we have used. In Section 5, we first relate multiple parameters to a program variable, and then group log messages to obtain message count vectors. In Section 6, we mainly present the invariant searching algorithm. We give a simple anomaly detection method in Section 7. Section 8 gives some experimental results on two large scale systems. Finally, we conclude the paper in Section 9.

2 Related Work

Recently, statistical machine learning and data mining techniques have shown great potential in tackling the scale and complexity of the challenges in monitoring and diagnosis of large scale systems. Several learning based approaches have been proposed to detect system failures or problems by statistically analyzing console logs, profiles, or system measurements. For example, Dickenson et al. [7] use classification techniques to group similar log sequences to a set of classes based on some string distance metrics. A human analyst examines one or several profiles from each class to determine whether the class represents an anomaly. Mirgorodskiy et al. [8] also use string distance metrics to categorize function-level traces, and identify outlier traces as anomalies that substantially differ from the others. Yuan et al. [9] first extract n-grams as features of system call sequences, and then use Support Vector Machine (SVM, a supervised classification algorithm) to classify traces based on the similarity of the traces of known problems. Xu et al. [5, 6] preprocess log messages to extract message count vectors as the log features, and detect anomalies using Principal Component Analysis (PCA). From the point of view of a human operator, the above statistical tools build models of a black box style, and they can hardly provide human operators with intuitive insights about abnormal jobs and anomalies. In [5, 6], the authors try to remedy the defect by learning a decision tree and to visualize the detection results. However, the decision tree is still somehow incomprehensible to human operators, because it does not directly relate to execution flow mechanisms. In this paper, we use program invariants to characterize the behavior of a system. Unlike the black box models, program invariants often provide intuitive interpretations of the detected anomalies.

Another set of algorithms [15, 16, 17] use Finite State Automaton (FSA) models to represent log sequences, which is more easily understood by operators. For example, SALSA [15] examines Hadoop logs to construct FSA models of the Datanode module and TaskTracker module. In the work of Cotroneo et al. [16], FSA models are first derived from the traces of Java Virtual Machine. Then, logs of unsuccessful jobs are compared with the inferred FSA models to detect anomalies. In [17], the authors also construct a FSA to characterize the normal system behaviors. A new trace is compared against the learned FSA to detect whether it is abnormal. However, these papers do not discuss interleaved logs which are prevalent in distributed systems. It is much more difficult to learn state machines from interleaved logs. Our analysis is based on message groups, which is not affected by the interleaving patterns.

Mining program invariants is a very important step in our approach. There are some research efforts related to this subject. Ernst et al. developed Daikon [10] to discover program invariants for supporting program evolution. Daikon can dynamically discover invariants at specific source code points by checking the values of all program variables in the scope. Jiang et al. proposed a search algorithm to infer likely invariants in distributed systems [12]. Rather than searching the invariants of program variables, their algorithm searches invariant pair-wise correlations between two flow intensities, such as traffic volume and CPU usage that are monitored in distributed systems. They also proposed an EM algorithm in [11], and extended their work to mine correlations among multiple flow intensities. In contrast with these methods, we mine invariant relationships among the counts of log message types, which present the characteristics of the program execution flow. In addition, we focus on sparse and integer invariants that can reveal the essential relations of the system execution logic and are easily understood by human operators.

3 The Approach

3.1 Invariants in textual logs

In general, a program invariant is a predicate that always holds the same value under different workloads or inputs. Program invariants can be defined from various aspects of a system, including system measurements (e.g. CPU and network utilization [11]) and program variables [10]. Besides the program variables and system measurements, program execution flows can also introduce invariants. With the assumption that log sequences provide enough information for the system execution paths, we can obtain invariants of program execution flows through analyzing log sequences. A simple example of program execution flow is shown in Fig. 1. At each stage of *A, B, C, D*, and *E*, the system prints a corresponding log message. We assume that there are multiple running instances that follow the execution flow shown in Figure 1. Even different instances may execute different branches and their produced logs may interleave together; the following equations should always be satisfied:

$$c(A) = c(B) = c(E) \quad (1)$$
$$c(B) = c(C) + c(D) \quad (2)$$

where $c(A)$, $c(B)$, $c(C)$, $c(D)$, $c(E)$ denote the number of log messages *A, B, C, D*, and *E* in the logs respectively. Each equation corresponds to a specific invariant of the program execution flow, and the validity of such invariants is not affected by the dynamics of the workloads, the difference of system inputs or the interleaving of multiple instances. In this paper, we call them *execution flow invariants*. There are mainly two reasons that we look at linear invariants among the counts of different type of log messages. First, linear invariants encode meaningful characteristics of system execution paths. They universally exist in many standalone or distributed systems. Second, an anomaly often manifests a different execution path from the normal ones. Therefore, a violation of such relations (invariants) means a program execution anomaly. Because log sequences record the underlying execution flow of the system components, we believe there are many such linear equations, i.e. invariants, among the log sequences. If we can automatically discover all such invariants from the collected historical log data, we can facilitate many system management tasks. For example,

- By checking whether a log sequence violates the invariants, we can detect system problems. As mentioned above, a violation of an invariant often means an anomaly in the program's execution.
- Each invariant contains a constraint or an attribute of a system component's execution flow. Based on the related execution flow properties of the broken invariants, system operators can find the potential

causes of failure.

- The invariants can help system operators to better understand the structure and behavior of a system.

3.2 Invariant as a Linear Equation

An invariant linear relationship can be presented as a linear equation. Given m different types of log messages, a linear relationship can be written as follows:

$$a_0 + a_1 x_1 + a_2 x_2 + \cdots + a_m x_m = 0$$

where x_j is the number of the log messages whose type index is j; $\theta = [a_0, a_1, a_2, \cdots, a_m]^T$ is the vector that represents the coefficients in the equation. So, an invariant can be represented by the vector θ. For example, the invariant of equation (2) can be represented by the vector $\theta = [0,0,1,-1,-1,0]^T$. Here, the message type indexes of A to E are 1 to 5 respectively. Obviously, independent vectors correspond to different linear relations, so represent different invariants. Given a group of log sequences $L_i, i = 1, \dots, n$, that are produced by past system executions, we count the number of every type of log messages in every log sequence, x_{ij}, $j = 1, \dots, m$. Here, x_{ij} is the number of log messages of the j^{th} log message type in the i^{th} log sequence. If none of those log sequences contains failures or problems, and all of log sequences satisfy the invariant, then we have the following linear equations:

$$a_0 + a_1 x_{i1} + \cdots + a_m x_{im} = 0, \forall i = 1, \dots, n \qquad (3)$$

Let us denote

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \ddots & x_{2m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix}$$

Then the formula (3) can be reformed as a matrix expression (4). That is to say, every invariant vector θ should be a solution of the equation:

$$\boldsymbol{X}\theta = 0 \qquad (4)$$

Formula (4) shows the principle characteristic of the invariants under the condition that all collected history logs do not contain any failure or problem. In practice, a few collected log sequences may contain failures or problems, which will make equation (4) not be precisely satisfied. We will discuss how to deal with such a problem in Section 6. Currently, we just focus on explaining the basic ideas and key concepts related to the execution flow invariant.

We derive two sub-spaces according to the matrix $\boldsymbol{X}$: the row space of matrix $\boldsymbol{X}$, which is the span of the row vectors of $\boldsymbol{X}$, and the null space of matrix $\boldsymbol{X}$, which is the orthogonal complement space to the row space. Formula (4) tells us that an invariant θ can be any vector in the null space of matrix $\boldsymbol{X}$. In this paper, we call the null space of matrix $\boldsymbol{X}$ the ***invariant space*** of the program. Each vector in the invariant space represents an execution flow invariant, and we call the vector in the invariant space the ***invariant vector***. On the other hand, any linear combination of the invariant vectors is also an invariant vector.

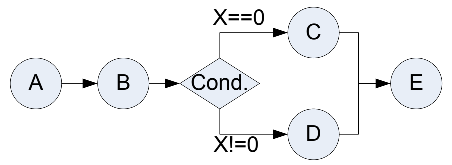


Figure 1. An execution flow example.

Sparseness: Although any vector in the invariant space represents an execution flow invariant, an arbitrary invariant vector with many non-zero coefficients often does not directly correspond to the meaningful work flow structures of a program and cannot be well understood by system operators.

Generally, the whole work flow of a program consists of a lot of elementary work flow structures, such as sequential structures, branched structures, and looping structures. The elementary work flow structures are often much simpler than the whole work flow structures and can be easily understood by system operators. As in the example shown in Figure 1, the sequential structure of A to B, the branch structure of B to C or D, and the joint structure of C or D to E are elementary work flow structures that compose the whole work flow. The invariants corresponding to the elementary work flow structures can usually give system operators intuitive interpretations of the system execution flow. For example, the invariant of $c(B) = c(C) + c(D)$ tells us that there may be a join or branch structure in the workflow.

Because the elementary work flow structures in the program are often quite simple, the invariants corresponding to such elementary work flow structures may involve only a few types of log messages. Therefore, compared to the number of all log message types, the number of message types involved in an elementary invariant is often very small, that is, the vector representations of such elementary invariants are often very sparse. Accordingly, the sparse vectors in the invariant space may correspond to elementary invariants, and general vectors in the invariant space may be linear

combinations of the elementary invariant vectors. For example, the equations (1) and (2) represent the elementary invariants that can directly reflect the elementary work flow structures in Figure 1. Each of them involves only two or three types of log messages. However, their linear combination, i.e. the invariant $c(A) + 3c(B) - 2c(E) - 2c(C) - 2c(D) = 0$, does not directly correspond to a concrete work flow structure in Figure 1 and is not easily understood by system operators.

In this paper, we assume that the number of log sequences is larger than the number of log message types. This assumption is reasonable because the number of log message types is constant and limited, while many logs can be collected while the system is running. If the dimension of the invariant space is r, then the dimension of the row space is $(m + 1 - r)$ because the dimension of the whole space is $(m + 1)$. Therefore, we can always find a vector with no more than $(m + 2 - r)$ non-zero coefficients in the invariant space. That is to say, the number of non-zero coefficients in a sparse invariant vector should be at most $(m + 1 - r)$, or it is not viewed as sparse because we can always find out an invariant vector with $(m + 2 - r)$ non-zero coefficients. We denote the upper bound of the number of non-zero coefficients for sparse invariant vectors as $K(\boldsymbol{X})$, and $K(\boldsymbol{X}) = m + 1 - r$. In real systems, the dimension of the row space is often quite small, so $K(\boldsymbol{X})$ is small too. For example, by investigating a lot of software systems, the authors of [5] observed that the effective dimensions of all row spaces are less than 4.

Compactness: For a set of invariant vectors, it is called a *redundant invariant vector set* if there is at least one invariant vector in the set that can be a linear combination of the other invariant vectors in the same set. On the other hand, if the set does not contain any invariant vector that can be a linear combination of the other invariant vectors in the same set, we call the set a *compact invariant vector set*. Because invariant vectors are essentially equivalent to invariants, it is natural to say that an invariant set is compact if its corresponding invariant vector set is compact, and vice versa. For example, the set $\{c(A) = c(B), c(A) = c(E), c(E) = c(B)\}$ is a redundant set, because the invariant $c(E) = c(B)$ can be deduced from the other two invariants in the set. On the other hand, the set $\{c(A) = c(B), c(A) = c(E), c(B) = c(C) + c(D)\}$ is a compact invariant set. Obviously, a redundant invariant set contains redundant information. If the dimension of the invariant space is r, there exists at most r different invariants satisfying that each of them cannot be a linear combination of the others. Therefore, for any compact invariant set C, the number of invariants in the set, i.e. $|C|$., is not larger than r.

Integer constraint: Another important property of the program execution flow invariants is that all coefficients are integer values. The reason is that all elementary work flow structures, such as sequence, branch, and join, can be interpreted by the invariant vectors whose elements are all integers. For example, the invariant vectors represented in equations (1) and (2) are all integer values, i.e. $[0,1,-1,0,0,0]^T$, $[0,0,1,0,0,-1]^T$ and $[0,0,1,-1,-1,0]^T$. At the same time, integer invariants are easily understood by human operators. In this paper, we aim to automatically mine the largest compact sparse integer invariant set of a program. In the remainder of this paper, the term "invariant" is used to refer to "sparse integer invariant" unless otherwise stated.

3.3 Practical Challenges

In real world systems, some collected historical log sequences contain failures or noise, i.e. they are abnormal log sequences. There also may be some partial log sequences, which are generally caused by inconsistent log data cuts from continuously running system (such as large-scale Internet Services). An invariant may be broken by these log sequences, because the abnormal execution flows are different from the normal execution flows. The results of this are that some of the equations in formula (3) may be not satisfied. With the assumption that failure logs and noise polluted logs only take up a small portion of the historical log sequences (e.g. <5%), we can find all invariants by searching the sparse resolutions of Equ. (4). This can be realized by minimizing the value of $\|\boldsymbol{X}\theta\|_{\mathbf{0}}$. Here, $\|\boldsymbol{X}\theta\|_{\mathbf{0}}$ equals the number of log sequences that violates the invariant θ.

Generally speaking, minimizing the value of $\|\boldsymbol{X}\theta\|_0$ is an NP-Hard problem [19]. To find a sparse invariant with k non-zero coefficients, the computational cost is about $O(C_m^k)$. Fortunately, in many systems, log messages may form some groups in which the log messages contain the same program variable value. Such groups usually represent meaningful workflows related to the specific program variable. For example, log messages containing an identical user request ID can form a group that represents the request handling execution flow in the program; there is a strong and stable correlation among log messages within the same group. On the other hand, inter-group log message types are often not obviously correlated. Furthermore, the number of log message types in each message group is usually much smaller than the total number of log message types. If we can divide all log messages into different groups properly and mine the invariants on different kinds of groups respectively, the search space of the algorithm can be largely reduced. There are some systems in which log messages do not contain such parameters. Just as prior work [5], our approach does not target these systems.

Even with the grouping strategy, the computational cost of invariant searching is still quite large. We try to further reduce the computational cost by introducing early termination and pruning strategies which will be discussed in Section 6.3.

3.4 Workflow of our approach

Figure 2 shows the overall framework of our approach, which consists of four steps: log parsing, log message grouping, invariant mining, and anomaly detection. We will provide further explanation in the corresponding sections.

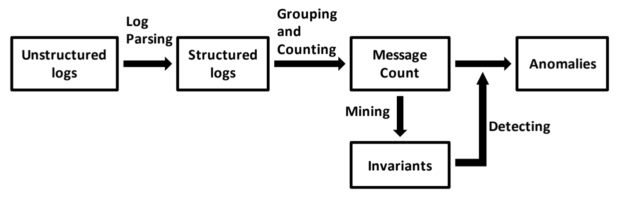


Figure 2. The overall framework of our approach.

Log parsing. In most systems, log messages are usually unstructured free-form text messages, and are difficult to be directly processed by a computer. In the log parsing step, we convert a log message from an unstructured text string to a tuple-form representation that consists of a timestamp, a message signature (i.e. a constant free form text string to represent a log message type), and a list of message parameter values.

Log message grouping and counting. Once parameter values and log message signatures are separated from all log messages, we first automatically discover whether a set of parameters correspond to the same program variable. Then, we group log messages that contain the same value of the same program variable together. For example, the log messages containing the same request ID value are grouped together. As mentioned above, dividing log messages into some close inner-related groups can largely reduce the computational cost. For each message group, we count the number of log messages for each message type to obtain a message count vector for further processing.

Invariant mining. Next, we try to find a compact sparse integer invariant set for each type of the log message groups. Message groups extracted according to the same program variable are considered as the same type of group. For example, the group of log messages with request ID #1# and the group of log messages with request ID #2# are the same type of message groups. In this paper, we combine a brute force searching algorithm and a greedy searching algorithm to make the searching process tractable.

Anomaly detection. We apply the obtained set of invariants to detect anomalies. A log sequence that violates an invariant is labeled as an anomaly.

4 Log Parsing

Each log message often contains two types of information: a free-form text string that is used to describe the semantic meaning of the recorded program event and parameters that are often used to identify the specific system object or to record some important states of the current system.

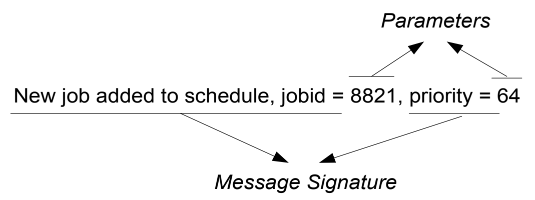

```
New job added to schedule, jobid = 8821, priority = 64
```

Figure 3. A log message contains two types of information: a message signature and parameters.

In general, log messages printed by the same log-print statement in the source code are the same type of messages because they all correspond to the same execution point and record the same kind of program events with the same semantic meaning. Different types of log messages are usually used to record different program events with different semantic meanings, and printed by different log-printed statements. Naturally, we can use the free-form text string in the log-print statement as a signature to represent the log message type. Therefore, a *message signature* corresponds to the constant content of all log messages that are printed by the same log-print statement. Parameter values are the recorded variable values in the log-print statement, and they may vary in different executions. For example, in Figure 3, the log message signature is the string "New job added to schedule, jobId =, priority =", and the parameter values are "8821" and "64".

The log parsing aims to extract message signatures and parameters from the original log messages. If the source code of the target program is available, the method proposed in [5] can automatically parse the log messages with a very high precision. However, in some systems, the source code is not available because it is usually stripped of programs' distribution packages. Therefore, it is still necessary to design a log parser that does not depend on any source code. In this paper, we use the algorithm that we have previously published [14] to extract message signatures and parameter values from log messages. It can achieve an accuracy of more than 95% [14]. Because the log parser is not the focus of this paper, we do not discuss the details of the algorithm.

Once the message signatures and parameter values are extracted from the original log messages, we can convert the unstructured log messages to their corresponding structured representations. For a log message m, we denote the extracted message signature as $K(m)$, the number of parameters as $PN(m)$, and the ith parameter's value as $PV(m, i)$. After message signature and parameter value extraction, each log message m with its time stamp $T(m)$ can be represented by a tuple $[T(m), K(m), PV(m,1), PV(m,2), \ldots, PV(m, PN(m))]$, we call such tuples the *tuple-form representations* of the log messages.

5 Log Message Grouping

The above log parsing procedure helps us to extract the message signature and parameter values for each log message. Each parameter is uniquely defined by a pair consisting of a message signature and a position index. Taking Figure 3 as an example, the pair ("*New job added to schedule, jobid=[],priority=[]* ", 1) defines a parameter, and its value is 8821 in the log messages. Note that a parameter is an abstract representation of a printed variable in the log-print statement, while a parameter value is the concrete value in a specific log message.

Developers often print out the same important program variable in multiple log-print statements. Therefore, multiple parameters may correspond to the same program variable, and we call these parameters cogenetic parameters. For example, a program variable, i.e. request ID, can appear as a parameter in different log message types, and these message types are often related to the execution flow of the request processing. Traditional log analysis tools heavily depend on application specific knowledge to determine whether a set of parameters correspond to the same program variable. However, in practice, most operators may not have enough knowledge about the implementation details. In this paper, we automatically determine whether two parameters are cogenetic. Our algorithm is based on the following observations:

- In a log bunch[1], if two parameters (e.g. P_a and P_b) either have the same value ranges (i.e. $V_r(P_a) = V_r(P_b)$), or one parameter's value range is a subset of the other's (e.g. $V_r(P_a) \subseteq V_r(P_b)$), then they are cogenetic (denoted as $P_a \cong P_b$). Here, the value range of a parameter ($V_r(P_a)$) is defined by the set of all distinct values of the parameter in the log bunch.
- Two parameters with a large joint set $V_r(P_a) \cap V_r(P_b)$ (namely the overlapped value range) will have a high probability to be cogenetic. For two parameters that are not cogenetic, they may have a few identical values in log messages by chance. However, if they have a lot of identical values, it is unlikely that it happens by chance. Therefore, a large overlapped value range often means that two parameters are likely cogenetic.
- The larger the length of each parameter value (i.e. the number of characters of the value in a log message) in the joint set $V_r(P_a) \cap V_r(P_b)$ is, the higher the probability the parameters are cogenetic. Intuitively, it will be more difficult for two parameter values with a large length to be identical by chance.

For each parameter, we first count its value range in each historical log bunch through scanning log messages one by one. Then, we check whether there are pair wise cogenetic relationships using Algorithm 1. For two parameters, if the size of the overlapped value range is larger than a threshold, and the minimal length of the parameter value is larger than 3 characters (Note that we use the text string form of the parameter values in the log messages for analysis), we can determine that the parameters are cogenetic. Finally, based on the obtained pair-wise cogenetic relationships, we can gather a set of parameters that are cogenetic into a parameter group. The detail of the algorithm is described in Algorithm 1.

According to the above algorithm, we obtain a set of parameter groups, with the parameters in each group being cogenetic. Intuitively speaking, each parameter

[1] In this paper, we can collect log messages several times as the target program runs under different workloads. At each time, one or more log files may be collected from distributed machines. The set of all the collected log messages at one time of collection are defined as *a log bunch*.

group corresponds to a program variable. We group log messages with the same program variable values together. Specifically, for each group of cogenetic parameters denoted as A, we group together the log messages that satisfy the following condition: the messages contain the parameters belonging to the specified parameter group A, and the parameter's values in the log messages are all the same.

Algorithm 1. Log parameter grouping

1. For each log parameter (defined by its message signature and its position index), we enumerate its value range (i.e. all distinct values) within each log bunch.
2. For every two parameters (e.g. P_a and P_b) that satisfy the following rules, we conclude that they are cogenetic parameters.
    - For every log bunch, we have $V_r(P_a) \subseteq V_r(P_b)$ or $V_r(P_a) \supseteq V_r(P_b)$.
    - $min(|V_r(P_a)|, |V_r(P_b)|) \geq 10$
    - Each value in $V_r(P_a)$ and $V_r(P_b)$ contains at least 3 characters.
3. We use the following rules to identify the group of cogenetic parameters:
    - If $P_a \cong P_b$ and $P_a \cong P_c$, we can conclude that P_a, P_b, and P_c are cogenetic parameters.

6 Invariant Mining

After the log message grouping step, we can obtain a set of log message groups for each program variable. Each message group describes a program execution path related to the program variable. Since the logging points are chosen by developers, log messages are often very important for problem diagnosis. We collect the log message groups corresponding to the same program variable together and discover their invariants as described in Algorithm 2. At first, for each log message group, we count the number of log messages for each log message type in the message group to obtain one message count vector. For all log messages groups that are related to the same program variable, we can extract a set of message count vectors. The message count vectors that correspond to the same program variable form the count matrix $\boldsymbol{X}$ (Eq. 4). Then, we need to identify the invariant space and the row space of $\boldsymbol{X}$ by using singular value decomposition and analysis. Next, we find the sparse invariant vectors in the invariant space. To find a sparse invariant vector with k non-zero coefficients with a small value of k (e.g, <5 in most cases), we can use a brute force search algorithm to obtain the optimal solution. However, when k is large, the brute force algorithm has to search in a huge searching space. In this case, we use a greedy algorithm [19] to obtain an invariant candidate. Finally, we validate the found invariants using the collected historical logs.

6.1 Estimate the invariant space

Once we have constructed the matrix $\boldsymbol{X}$ from the collected historical logs, we can estimate the invariant space by singular value decomposition (SVD) operation.

Instead of the energy ratio, we use the support ratio as a criterion to determine the invariant space (and, at the same time, the row space). It can directly measure the matching degree between the collected logs and the invariants. For an invariant, the support ratio is defined as the percentage of the log message groups that do not break the invariant. Specifically, we first use SVD to obtain the right-singular vectors. After that, we evaluate the right-singular vectors one by one in increasing order of singular values to check whether they are a part of the invariant space. For a right-singular vector v_i, if there are more than 98% log message groups satisfying the condition $|X_j v_i| < \epsilon$, we treat v_i as a validated invariant. Otherwise, it is an invalidated invariant. Here X_j is a message count vector of the message group j, ϵ is a threshold. The right-singular vector with the smallest singular value is evaluated first. Then, the vector with the second smallest singular value is evaluated, and so on. If a singular vector is verified as an invalidated invariant, the evaluation process is terminated. The invariant space is a span of all right-singular vectors that have been validated during this process. In our implementation, the threshold ϵ is selected as 0.5 ($\approx \sqrt{4}/4$) because most of our invariants at most contain 4 non-zero coefficients.

6.2 Invariant searching

In this section, we introduce an invariant searching algorithm which aims to find a compact set of program invariants based on a log message count matrix $\boldsymbol{X}$. Because we have little knowledge about the relationship between different log message types, we try any hypotheses of non-zero coefficients in different dimensions to construct a potential sparse invariant, and then continue to validate whether it fits with the historical log data.

Specifically, we define an invariant hypothesis as its non-zero coefficient pattern $\{p_j, j = 1,2, \cdots, k\}$, where p_j is the index of the non-zero coefficient of the invariant hypothesis, and $0 \leq p_j < p_{j+1} \leq m$. For any non-zero coefficient pattern, we check whether there is an invariant, i.e. $\{a_{p_j}, j = 1,2, \cdots, k\}$. There are two steps. At first, we try to obtain an invariant candidate $\hat{\theta}$ that satisfies the given non-zero coefficient pattern and minimizes the value of $\|\widehat{\boldsymbol{X}}\hat{\theta}\|_{\boldsymbol{0}}$, namely $\hat{\theta} = \text{argmin}_{\theta}(\|\widehat{\boldsymbol{X}}\theta\|_{\boldsymbol{0}})$. Here, $\widehat{\boldsymbol{X}}$ is a matrix that contains only k column vectors of matrix $\boldsymbol{X}$ whose column indexes are $\{p_j, j = 1,2, \cdots, k\}$. We ignore other columns in X for constructing $\widehat{\boldsymbol{X}}$ because those columns correspond to zero coefficients of the invariant vectors. Because an optimization operation over zero norm is often not tractable, we estimate $\hat{\theta}$ through $\hat{\theta} = \text{argmin}_{\theta}(\|\widehat{\boldsymbol{X}}\theta\|_{\boldsymbol{2}})$. The coefficients of the estimated $\hat{\theta}$ are often within the range of $(-1.0, 1.0)$. In order to obtain integer invariant candidates, we scale up $\hat{\theta}$ to make its minimal non-zero coefficient equal to an integer l, and round other non-zero coefficients to an integer accordingly. In this paper, we set $l = 1,2, \cdots, p$ respectively. Therefore, we obtain p integer invariant candidates. Then, we verify each of them by checking its support ratio based on the log message groups. If there is an invariant whose support ratio is larger than γ, we set it as a validated invariant, otherwise, we conclude that there is no invariant that satisfies the non-zero coefficients pattern $\{p_j, j = 1,2, \cdots, k\}$. Here, γ is a user defined threshold, which is set as 98% in our experiments. In our implementation, we can handle all cases that we have studied by selecting $p = 3$. A large value of p is often not necessary, because most complex invariants are linear combinations of simple local invariants.

Algorithm 2. Mining Invariants

1. For all message groups related to a specific parameter group, we construct the matrix X using their message count vectors, and estimate the dimension of the invariant space (denoted as r).
2. We use a brute force algorithm to search invariants that contains k non-zero coefficients, where k increases from 1 to 5 in turn. The algorithm exits when one of following conditions is satisfied:
    - r independent invariants have been obtained.
    - $k > (m - r + 1)$
3. If $(m - r + 1) > 5$ and no early terminate condition has be satisfied, we use a greedy algorithm [19] to find potential invariants for *k >5*.

6.3 Computational Cost and Scalability

In general, it is an NP-Hard problem to find a sparse invariant vector. The computational complexity of the above search algorithm is about $O(\sum_{i=1}^{m-r+1} C_m^i)$. Although it has been largely reduced from the computational cost of full search space (i.e. $O(\sum_{i=1}^{m} C_m^i)$), it is still not a trivial task if the number of dimensions of matrix $\boldsymbol{X}$'s row space (i.e. $m - r + 1$) is large. Fortunately, in real world systems, the dimensions of the row spaces are often very small, which helps us to avoid the problem of combinatorial explosion. For example, in Table 1, we list the row space dimensions of different types of log message groups. Many of them are not larger than 4. Therefore, the computational cost can usually be controlled below $O(\sum_{i=1}^{4} C_m^i)$.

In addition, most real world sparse invariants often only contain 2 or 3 non-zero coefficients. Because we can obtain at most r independent invariants, we do not need to search the combinations of 5 or more non-zero coefficients if we have obtained r independent invariants when k<5. For example, the 4th row of Table 3 is such a case. This allows us to terminate the search process early, and to reduce the computational cost.

Table 1. Low dimensionality of row space

Message group of related object identifier	m	$\boldsymbol{m - r + 1}$
Hadoop logs with MapTask ID	7	3
Hadoop logs with ReduceTask ID	3	2
Hadoop logs with MapTask Attempt ID	28	4
Hadoop logs with ReduceTask Attempt ID	25	6
Hadoop logs with JVM ID	7	2

Table 2. Reduce computational cost

Message group of related object identifier	Original search space	Result search space
Hadoop logs with MapTask ID	63	37
Hadoop logs with ReduceTask ID	6	6
Hadoop logs with MapTask Attempt ID	24157	3310
Hadoop logs with ReduceTask Attempt ID	15275	730
Hadoop logs with JVM ID	28	16

Furthermore, we can reduce the computational cost by skipping the searching on some hypothesis candidates. As discussed in Section 3, any linear combination of invariants is also an invariant. Therefore, we need not search the invariants that can be a linear combination of the detected ones. Then, the search space can be largely reduced by skipping the search on such combinations. At the same time, the skipping strategy also guarantees the compactness of our discovered invariants. Table 2 shows the effectiveness of our early termination and skipping strategy. The numbers of hypotheses in the original search space (i.e. $O(\sum_{i=1}^{m-r+1} C_m^i)$) are listed in the second column. The third column contains the size of the search space after applying the early termination and the skipping strategy. By comparing the search space size in the two columns, we can find that the early termination and the skipping strategy largely reduce the search space, especially for the message groups with high dimension values.

In our implementation, we only search the invariant hypotheses up to 5 non-zero coefficients. If no early termination condition is met, we then find potential invariants by using a greedy algorithm [19] on the message types that do not appear in the existing invariants. However, the greedy algorithm cannot guarantee to find all invariants in logs. The overall algorithm is presented in Algorithm 2.

From the view of scalability, there are a huge amount of log messages in a large scale system with thousands of machines. Therefore, the row number n of matrix X is often very large. Directly applying SVD on matrix X is often not scalable. Fortunately, the number of message types (i.e. m) is usually limited, and it does not increase as the system scales up. We can replace the SVD operation by an Eigen Value Decomposition (EVD) operation to calculate the right-singular vectors, because $\boldsymbol{X} = U\Lambda V^T \Rightarrow \sum = \boldsymbol{X}^T\boldsymbol{X} = V\Lambda^T\Lambda V^T$. Here, matrix $\sum = \boldsymbol{X}^T\boldsymbol{X}$ is a $m \times m$ matrix. It can be easily calculated by a MapReduce-like distributed program. Similarly, we can also use an EVD operation to estimate $\hat{\theta}$ based on a matrix $\widehat{\sum} = \widehat{\boldsymbol{X}}^T\widehat{\boldsymbol{X}}$ for each invariant hypothesis**.** The matrix $\widehat{\sum}$ can directly be calculated from the matrix $\sum$. At the same time, the support ratio counting procedure can also be easily performed in a distributed manner. Therefore, our algorithm can be easily scaled up. In addition, most program invariants do not depend on the scale of the system. We can learn invariants from the logs of a small scale system deployment, and then use these invariants to detect problems of a large scale deployment.

7 Problem Detection

Once the program invariants are discovered, it is straightforward to detect problems from console logs. For a new input console log, we first convert the unstructured log messages to tuple-form representations using the log parser, and then group log messages and calculate a count vector for each message group. After that, we check every message count vector with its related learned invariants. The message group whose message count vector violates any one of its related invariants is considered as an anomaly.

The automatically mined invariants by our approach reflect the elementary execution flows. These program invariants often provide intuitive and meaningful information to human operators, and help them to locate problems on the fine granularity. Therefore, we relate each detected anomaly with the invariants that it breaks so as to provide insight cues for problem diagnosis. Operators can check which invariants are broken by an anomaly, and how many anomalies are raised by the violation of a specific invariant.

8 Case Study and Comparision

In this section, we evaluate the proposed approach through case studies on two typical distributed computing systems: Hadoop and CloudDB, a structured data storage service developed by Microsoft. We first set up a testing environment and collect console logs. And then, we begin our experiments of anomaly detection on these two systems. The detection results are presented and analyzed in the following subsections. Unlike CloudDB, Hadoop is a publicly available open-source project. The results on Hadoop are easy to be verified and reproduced by third parties. Therefore, we give more details about the results on Hadoop.

8.1 Case Study on Hadoop

Test Environment Setup: Hadoop [18] is a well-known open-source implementation of Google's MapReduce framework and distributed file system (GFS). It enables distributed computing of large scale, data-intensive and stage-based parallel applications.

Our test bed of Hadoop (version 0.19) contains 15 slave workstations (with 3 different hardware configurations) and a master workstation, and all these machines are connected to the same 1G Ethernet switch. We run different Hadoop jobs including some simple sample applications, such as WordCount and Sort, on the test bed. The WordCount job counts the word frequency in some random generated input text files, and the Sort job sorts the numbers in the input files. During the running of these jobs, we randomly run some resource intensive programs (e.g. CPUEater) on the slave machines to compete for CPU, memory and network resources with Hadoop jobs. Rather than active error injection, we hope such intensive resource competition can expose inherent bugs in Hadoop. We collect the produced log files of these jobs 4 times at different time points. Each time, we put the collected log files into a single file folder. The log data is not uniformly distributed in these folders. The smallest folder contains about 116 megabytes, and the largest folder contains about 1.3 gigabytes. The log messages in each folder are considered as a log bunch. There are totally about 24 million lines of log messages.

Results of Parameter Grouping: Our parameter grouping algorithm identifies several parameter groups as meaningful program variables. By manually checking the log messages and the parameters, we find that they corresponded to the following meaningful program variables: Map/Reduce Task ID, Map/Reduce Task Attempt ID, Block ID, and JVM ID, Storage ID, IP address and port, and write data size of task shuffling. These variables include both object identifiers (such as Task ID) and system states (such as IP address and Port). It is interesting that the packet size during shuffling operations is also detected as a parameter group. We discover one invariant from its related message group, and learn that the number of MAPRED_SHUFFLE operations is equal to the number of messages of "Sent out bytes for reduce:## from map:## given from with (#,#)".

Table 3. Invariants found in Hadoop logs

Message groups of related object identifiers	Invariants (≤3 coef.)	Invariants (≥4 coef.)
Hadoop logs with MapTask ID	3	0
Hadoop logs with ReduceTask ID	1	0
Hadoop logs with MapTask Attempt ID	21	3
Hadoop logs with ReduceTask Attempt ID	17	0
Hadoop logs with Data Block ID	9	0
Hadoop logs with JVM ID	5	0
Hadoop Logs with Storage ID	3	0
Logs with IP/port	4	0
Logs with task write packet size	1	0

Learned Invariants: In the Hadoop experiments, we discover 67 invariants in total. 64 of them only contain at most three non-zero coefficients, and 3 invariants have 4 non-zero coefficients. Table 3 shows the numbers of the learned invariants for different program variables. To validate our learned invariants, we manually verify our learned program invariants by carefully studying the Hadoop source code, the documents on MapReduce, and the sample logs. By comparing with the real program work flows, we find that our discovered invariants correctly describe the inherent linear relationships in the work flow. No false positive invariant is found. To vividly illustrate our discovered invariants, we present an example - the learned ternary invariant of the MapTask log group. The invariant equation is $c(L_{113}) + c(L_{114}) = c(L_{90})$, where L_{113}, L_{114}, and L_{90} are the log message types of "*Choosing data-local task ##*", "*Choosing rack-local task ##*", and "*Adding task '##' to tip ##, for tracker '##'*" respectively. In our test environment, all 16 machines are connected to a single Ethernet switch, and they are configured as one rack. Therefore, for each MapTask, it selects its data source from either local disc or local rack. The above invariant correctly reflects this property because the equation shows that each "*Adding task*" corresponds to either a "*data-local*" or a "*rack-local*". This proves our claim in Section 3 that invariants encode the properties of work flow structures.

Table 4. Detected true problems in Hadoop

Anomaly Description	PCA based Method	Our Method
Tasks fail due to heart beat lost.	397	779
A killed task continued to be in RUNNING state in both the JobTracker and that TaskTracker for ever	730	1133
Ask more than one node to replicate the same block to a single node simultaneously	26	26
Write a block already existed	25	25
Task JVM hang	204	204
Swap a JVM, but mark it as unknown.	87	87
Swap a JVM, and delete it immediately	211	211
Try to delete a data block when it is opened by a client	3	6
JVM inconsistent state	73	416
The pollForTaskWithClosedJob call from a Jobtracker to a task tracker times out when a job completes.	3	3

Anomaly Detection: We use the learned invariants to detect anomalies by checking whether a log sequence breaks a program invariant. By manually checking these detected anomalies, we find that there are 10 types of different execution anomalies, which are listed in Table 4. Note that each anomaly in Table 4 corresponds to a specific pattern corresponding to a certain set of violated invariants, and its description is manually labeled by our carefully studying the source code and documents of Hadoop. Many of them are caused by the loss of the heart beat message from Tasktracker to Jobtracker. Our method also detects some subtle anomalies in Hadoop. For example (the 4th row of Table 4), we detect that Hadoop DFS has a bug that asks more than one node to send the same data block to a single node for data replication. This problem is detected because it violates a learned invariant of "count('Receiving block ##') = count('Deleting block file ##')". A node receives more blocks than it finally deletes, and some duplicated received blocks are dropped.

At the same time, we find that our approach can well handle the problems that cause the confusion in the traditional keyword based log analysis tools. Here is one typical example. In Hadoop, TaskTracker often logs many non-relevant logs at info level for DiskChecker$DiskErrorException. According to Apache issue tracking HADOOP-4936, this happens when the map task has not created an output file, and it does not indicate a running anomaly. Traditional keyword-based tools may detect these logs as anomalies, because they find the word *Exception*. This confused many users. Unlike keyword-based tools, our approach can avoid generating such false positives.

Table 5. False positives

False Positive Description	PCA Method	Our Method
Killed speculative tasks	585	1777
Job cleanup and job setup tasks	323	778
The data block replica of Java execution file	56	0
Unknown Reason	499	0

Just like all unsupervised learning algorithms, our approach does detect some false positives. As shown in Table 5, we detect two types of false positives. Hadoop has a scheduling strategy to run a small number of speculative tasks. Therefore, there may be two running instances of the same task at the same time. If one task instance finishes, the other task instance will be killed no matter which stage the task instance is running at. Some log groups produced by the killed speculative task instances are detected as anomalies by our approach, because their behaviors are largely different from normal tasks. The other false positives come from job cleanup and setup tasks. Hadoop schedules two tasks to handle job setup and cleanup related operations. Since these two tasks, i.e. job setup task and job cleanup task, print out the same type of log messages as map tasks. Many users are confused by these logs. Because their behaviors are quite different from the normal worker map tasks, our approach also detects them as anomalies.

Comparison with the PCA Based Algorithm: We compared our approach with the PCA based algorithm of [5]. Because our running environment, work load characteristics, and Hadoop version are different from the experiments in [5], we cannot directly compare our results with theirs. We implement their algorithm and test it on our data set. From Table 4 and Table 5, we can find that both algorithms can detect the same types of anomalies, which is reasonable because both approaches utilize the inherent linear characteristics of the console logs. In some cases, our approach can detect more anomalies than the PCA based approach. If a set of log messages appear in almost all log message groups, the PCA based algorithm will ignore them by giving a very small TF/IDF weight. Therefore, the PCA based algorithm cannot detect the anomalies exposed as abnormal relationships among the log message types. For example, in a case of "*JVM inconsistent state*" (refer to the 10th row of Table 4), our algorithm detects the anomaly because the message "*Removed completed task ## from*" abnormally appears twice for the same task instance (i.e. breaking an invariant of one message for each task). However, the PCA based algorithm cannot detect these anomalies because it ignores the message. On the whole, in our test data, our approach can detect all the anomalies that can be detected by the PCA based method.

Unlike the PCA based approach, our invariant based approach can give human operators intuitive insight of an anomaly, and help them to locate anomalies in finer granularity. For example, the fact that an anomaly of "*a task JVM hang*" (refer to the 5th row of Table 4) breaks the invariant of "count('JVM spawned') = count('JVM exited')" can give operators a very useful cue to the understanding of the anomaly: a JVM spawned but does not exit, which may indicate the JVM is hung. At the same time, because the anomaly does not break "count('JVM spawned') = count('JVM with ID:# given task:#')", we can conclude that the JVM got hung after it was assigned a MapReduce task. However, the PCA based approach can only tell operators that the feature vector of the anomaly is far away from the normal feature space, i.e. the residential value of the feature vector is larger than a certain threshold value. This can hardly help operators to diagnose the problems, and they have to check the original log sequence carefully for problem analysis. In the PCA based approach, the decision tree technique makes its decision based on the count of one type of log messages at each step of the decision tree [5]. In contrast, our invariant based approach utilizes the numerical relationships among different types of log messages.

The 4th row of Table 5 shows another advantage of our approach. In order to rapidly distribute the Java executable files (e.g. JAR file) of jobs, Hadoop sets the replica number of these files according to the number of slave machines. For example, in our test environment, the replica number of a job JAR file is set as 15. The PCA based algorithm detects them as anomalies, because 15 is far from the normal replica number (e.g. 3 in most systems) of the data block. Our approach does not detect them as anomalies because their work flows do not break any invariant. There are some other false positive cases (refer to the 5th row of Table 5), e.g. some log groups of successful tasks, in the results of PCA based algorithm. We speculate that these false positives may be caused by the different characteristics of work load (WordCount and Sort are different), but currently, we do not know the exact reason. It seems that the PCA based method is more sensitive to the workload and environment. Our algorithm is much more robust. Furthermore, our algorithm only detects two types of false positives, while the PCA based method detects more than 4 types (we believe that the unknown reason false positives belong to many different types.). We argue that, rather than the number of false positives, the number of false positive types is more important for a problem detection tool. In fact, when the tool detects an anomaly, if a human operator marks it as a false positive, the tool can automatically suppress to pop up false positives of the same type. Therefore, a tool with few types of false positives can reduce the operator's workload.

8.2 Case Study on CloudDB

MS CloudDB is a structured data storage service developed for internal usage in Microsoft. It can scale up to tens of thousands of servers and runs on commodity hardware. It is capable of auto-failover, auto-load balancing, and auto-partitioning. In our experiments, we use the log messages of Fabric and CASNode levels, which implement the protocols and life cycles of distributed storage nodes, to learn invariants and detect potential anomalies. About 12 million log messages are analyzed in the experiment. We first manually construct some work flow models based on the documents provided by the product group. Due to the insufficiency of documents, not all work flows involved in these two levels are constructed. Then, we compare the invariants automatically mined by our approach (266 invariants are learned in this experiment) with the manually constructed work flow models. The mined invariants not only correctly reflect the real work flows, but also help us to find out some mistakes in the manually constructed work flow models that are caused by the misunderstanding of some content in the documents.

Table 6. Detected anomalies in CloudDB

Anomaly Description	PCA Method	Our Method
Data store operation finished without client response	0	2
Service message lost	8	8
Refresh config message lost	0	2
LookupTableUpdate message lost	0	1
AddReplicaCompleted message lost	1	8
Fail to close channel	2	67
No response for an introduce request	0	2
Send depart message exception	0	2
Add primary failed	0	2

After that, we also use the learned invariants to perform anomaly detection. Table 6 summarizes the detected anomalies in the experiment. By comparing Table 6 and Table 4, we can obtain a similar conclusion as that in Section 8.1 about the performances of the two methods. As mentioned in Section 8.1, the PCA based algorithm fails to detect lots of anomalies because it gives a very small TF/IDF weight to each routine message.

9 Conclusions

In this paper, we propose a general approach to detecting system anomalies through the analysis of console logs. Because the log messages are usually free form text strings that can hardly be analyzed directly, we first convert unstructured log messages to structured logs in tuple-form representations. The parameters that represent the same program variable are classified into a parameter group. We identify parameter groups by analyzing the relationships among the value ranges of the parameters. Then, according to these parameter groups, we classify the log messages to log message groups, and construct message count vectors. After that, we mine the sparse, integer valued invariants from the message count vectors. The mined invariants can reflect the elementary work flow structures in the program. They have physical meanings, and can be easily understood by operators. Finally, we use the discovered invariants to detect anomalies in system logs. Experiments on large scale systems such as Hadoop and CloudDB have shown that our algorithm can detect numerous real problems with high accuracy, which is comparable with the state of art approach [5]. In particular, our approach can detect anomalies with finer granularity and provide human operators with insight cues for problem diagnosis. We believe that this approach can be a powerful tool for system monitoring, problem detection, and management.

Acknowledgements

We thank Wei XU from UC Berkley for his helpful discussion, and our colleagues, Zheng Zhang, Lidong Zhou and Zhenyu Guo, for their valuable comments on the project. We also thank our paper shepherd, Emin Gün Sirer, and the anonymous reviewers for their insightful comments and advice.

10 References

[1] J. P. Rouillard, *Real-time Log File Analysis Using the Simple Event Correlator (SEC),* In Proc. of the 18th Usenix LISA'04, Nov. 14-19, 2004.

[2] J. E. Prewett, *Analyzing Cluster Log Files Using Logsurfer*, In Proc. of Annual Conference on Linux Clusters, 2003.

[3] S. E. Hansen, and E. T. Atkins, *Automated System Monitoring and Notification with Swatch*, In Proc. of the 7th Usenix LISA'93, 1993.

[4] A. Oliner and J. Stearley. *What supercomputers say: A Study of Five System Logs*. In Proc. of IEEE DSN'07, 2007.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, *Detecting Large-Scale System Problems by Mining Console Logs*, In Proc. of ACM SIGOPS SOSP'09, Big Sky, MT, Oct. 11-14, 2009.

[6] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, *Mining Console Logs for Large-Scale System Problem Detection*, In Proc. of SysML, Dec. 2008.

[7] W. Dickinson, D. Leon, and A. Podgurski, *Finding Failures by Cluster Analysis of Execution Profiles*, In Proc. of ICSE, May 2001.

[8] A.V. Mirgorodskiy, N. Maruyama, and B.P. Miller, *Problem Diagnosis in Large-Scale Computing Environments*, In Proc. of the ACM/IEEE SC 2006 Conference, Nov. 2006.

[9] 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 Proc. of EuroSys'06, Apr. 2006.

[10]M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, In IEEE Trans. on Software Engineering, pp. 99-123, Vol.27, No.2, Feb. 2001.

[11]H. Chen, H. Cheng, G. Jiang, and K. Yoshihira, *Exploiting Local and Global Invariants for the Management of Large Scale Information Systems*, In Proc. of ICDM'08, Pisa, Italy, Dec. 2008.

[12]G. Jiang, H. Chen, and K. Yoshihira, *Efficient and Scalable Algorithms for Inferring Likely Invariants in Distributed Systems*, In IEEE Trans. on Knowledge and Data Engineering, pp. 1508-1523, Vol.19, No. 11, Nov. 2007.

[13]H. Chen, G. Jiang, C. Ungureanu, and K. Yoshihira, *Failure Detection and Localization in Component Based Systems by Online Tracking*, In Proc. of SIGKDD, pp. 750-755, 2005.

[14]Q. Fu, J.-G. Lou, Y. Wang, and J. LI*, Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis*, In Proc. of ICDM, Florida, Dec. 2009.

[15] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasim-han, *SALSA: Analyzing Logs as State Machines*, In Proc. of WASL, Dec. 2008.

[16] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, *Investigation of Failure Causes in Workload Driven Reliability Testing*, In Proc. of the 4th Workshop on Software Quality Assurance, Sep. 2007.

[17] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira. Multi-*resolution Abnormal Trace Detection Using Varied-length N-grams and Automata*, In Proc. of ICAC, Jun. 2005.

[18] Hadoop. http://hadoop.apache.org/core.

[19] D. Donoho, Y. Tsaig, I. Drori, and J. Starck, *Sparse Solutions of Underdetermined Linear Equations by Stagewise Orthogonal Matching Pursuit*, Technical Report of Department of Statistics, Standford, TR2006-02, 2006.

Wide-area Network Acceleration for the Developing World

Sunghwan Ihm[1], KyoungSoo Park[†2], and Vivek S. Pai[1]

[1]*Department of Computer Science, Princeton University*
[2]*Department of Electrical Engineering, KAIST*

Abstract

Wide-area network (WAN) accelerators operate by compressing redundant network traffic from point-to-point communications, enabling higher effective bandwidth. Unfortunately, while network bandwidth is scarce and expensive in the developing world, current WAN accelerators are designed for enterprise use, and are a poor fit in these environments.

We present Wanax, a WAN accelerator designed for developing-world deployments. It uses a novel multi-resolution chunking (MRC) scheme that provides high compression rates and high disk performance for a variety of content, while using much less memory than existing approaches. Wanax exploits the design of MRC to perform intelligent load shedding to maximize throughput when running on resource-limited shared platforms. Finally, Wanax exploits the mesh network environments being deployed in the developing world, instead of just the star topologies common in enterprise branch offices.

1 Introduction

While low-cost laptops may soon improve computer access for the developing world, their widespread deployment will increase the demands on local networking infrastructure. Locally caching static Web content can alleviate some of this demand, but this approach has limits on its effectiveness, especially in smaller environments.

We propose to augment these caches with integrated wide area network (WAN) accelerators that have been specifically designed to operate in developing-world environments. WAN accelerators are deployed near edge routers, and work by compressing redundant traffic destined to locations with other WAN accelerators. To compress traffic, the accelerators break the data stream into smaller chunks, store these chunks at each accelerator, and then replace future instances of this data with reference to the cached chunks. By passing references to the chunks rather than the full data, the accelerator compresses the data stream.

Current WAN accelerators are not well-suited for the developing world. While they typically require server-class machines with a set of fast disks and a large pool of dedicated memory, the average school targeted by the One Laptop Per Child (OLPC) project will have 100 laptops in the price range of US $100-$200 each, for a total cost of $10K-$20K [22]. Requiring special server-class hardware for WAN acceleration alone could increase deployment cost. Other options would be to share the machine with other services (e.g, mail servers, Web servers, and proxies) or to use cheap, laptop-class hardware, both of which would reduce the RAM and disk available to the WAN accelerator. In addition, existing designs cannot exploit the mesh network environments being deployed in the developing world, limiting their potential utility.

[†]Work partly done while at University of Pittsburgh.

We have developed a new WAN accelerator, Wanax, that is designed to meet these challenges in the developing world. Our technical contributions are the followings: (1) a novel multi-resolution chunking (MRC) technique, which provides high compression rates and high disk performance across workloads while having a small memory footprint; (2) an intelligent load shedding technique that exploits MRC to maximize effective bandwidth by adjusting disk and WAN usage as appropriate; and (3) a mesh peering protocol that exploits higher-speed local peers when possible, instead of fetching only over slow WAN links. The combination of these design techniques makes it possible to achieve high effective bandwidth even with resource-limited shared machines.

The rest of this paper is organized as follows: §2 provides background on WAN accelerators and new challenges in the developing world. §3 describes the design of Wanax, and we show the trace-based simulation analysis in §4. In § 5, we detail the prototype implementation, and §6 presents the experimental results. Finally, we discuss related work in §7, and conclude in §8.

2 Background and Motivation

Our goal is to improve Internet access in the developing world using WAN accelerators designed to use low-end hardware. We primarily focus on increasing the *effective bandwidth* (or throughput) of the expensive, low-bandwidth WAN link in the region. We first provide a brief introduction to WAN accelerators, and then discuss the specific problems.

2.1 WAN Accelerators

Content Fingerprinting Content fingerprinting (CF) forms the basis for WAN acceleration, since it provides a position-independent and history-independent technique for breaking a stream of data into smaller pieces, or chunks, based only on their content.

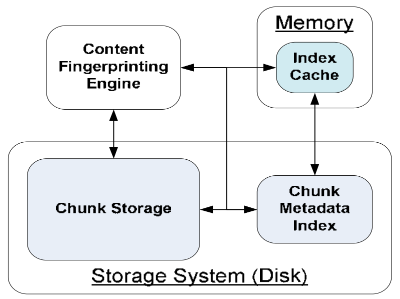


Figure 1: WAN Accelerator Architecture

While early systems used Manber's anchor technique to determine chunk boundaries [18], Rabin's fingerprinting technique is now widely used for its efficiency and flexibility [29]. It continuously generates integer values, or fingerprints, over a sliding window (e.g., 48 bytes) of a byte stream. When a fingerprint matches a specified global constant K, that region constitutes a chunk boundary. The average chunk size can be controlled with a parameter n, that defines how many low-order bits of K are used to determine chunk boundaries. In the average case, the expected chunk size is 2^n bytes. To prevent chunks from being too large or too small, minimum and maximum chunk sizes can be specified as well. Since Rabin fingerprinting determines chunk boundaries by content, rather than offset, localized changes in the data stream only affect chunks that are near the changes.

Once a stream has been chunked, the WAN accelerator can cache the chunks and pass references to previously cached chunks, regardless of their origin. As a result, WAN accelerators can compress within a stream, across streams, and even across files and protocols.

Performance Trade-offs Figure 1 depicts the general architecture of modern WAN accelerators. Chunk data is stored on disk due to cost and capacity, but an index of chunk metadata is partially or completely kept in memory to avoid disk accesses. Memory also serves as a cache for chunk data, to reduce disk access for commonly-used content.

The performance of WAN accelerators is mainly determined by three factors - (1) *compression rate*, (2) *disk performance*, and (3) *memory pressure*. Compression rate refers to the fraction of the original data actually gets sent, and reflects network bandwidth savings by receiver-side caching. Disk performance determines the cached chunk access time (seek time) while memory pressure affects the efficiency of the chunk index and in-memory caching. These three factors affect the total latency, which is the time to reconstruct and deliver the original data. Delivering high effective bandwidth requires reducing the total latency – having *high compression*, *low disk seeks*, and *low memory pressure* simultaneously.

Chunk size directly impacts all three factors, and consequently the effective bandwidth as well. Small chunks can lead to better compression if changes are fine-grained, such as a word being changed in a paragraph. Only the chunk containing the word is modified, and the rest of the paragraph can be compressed. However, for the same storage size, smaller chunks create more total chunks, increasing the metadata index size, and increasing the memory pressure and disk seeks. Large chunks yield fewer chunks in total, reducing memory pressure from indexing and providing better disk usage since each read can provide more data. Large chunks, however, can miss fine-grained changes, leading to lower compression. No chunk size is standard in systems that use content fingerprinting – for example, VBWC [30] uses a 2KB chunk size, LBFS [21] uses 8KB, and Shark [5] uses 16KB.

2.2 Developing World Challenges

Our target environment, schools in the developing world, is very different from enterprise branch offices, the typical candidate for WAN accelerators.

Limited RAM Due to cost, schools want a *shared machine* or a cheap laptop with limited RAM running the WAN accelerator and other services, instead of using a dedicated server appliance. Also, school children may want to access any content on the Internet, rather than just a smaller set of work-related documents in the enterprise environment. This *larger working set* requires more disk storage, more chunks, and more metadata entries, increasing memory pressure.

Poor Disk Performance While disk capacity is cheap and large (1TB SATA per $100), disk seek performance is still limited and is often the bottleneck. Modern desktop drives typically perform roughly 100 seeks/second, but cheaper laptop/external drives we may expect in the developing world are even slower, and are much slower than the high-RPM SCSI disks commercial WAN accelerators use. Also, the larger working set and other services sharing the disks further increase the disk load.

Low Compression Rate To handle poor disk performance in the developing world, one choice is to use large chunks to reduce the number of disk accesses, but this reduces the compression rate, limiting bandwidth gains.

Mesh Topology Enterprise branch offices typically communicate with a central office in a star topology, whereas many schools in a local region may prefer to get content from each other over cheaper local links rather than over the WAN link. Current WAN accelerators are not designed to exploit this opportunity.

3 Wanax Design

Motivated by the challenges in the developing world, we design Wanax around four goals - (1) maximize compression, (2) minimize disk seeks, (3) minimize memory pressure, and (4) exploit local resources.

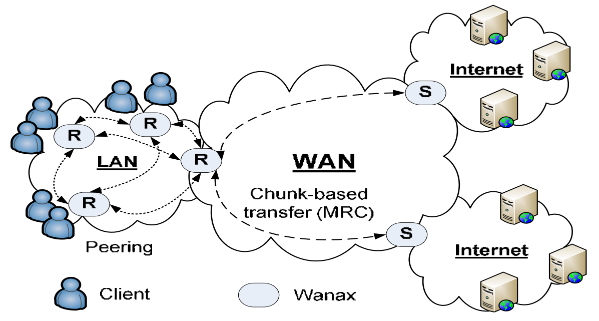


Figure 2: Wanax System Overview

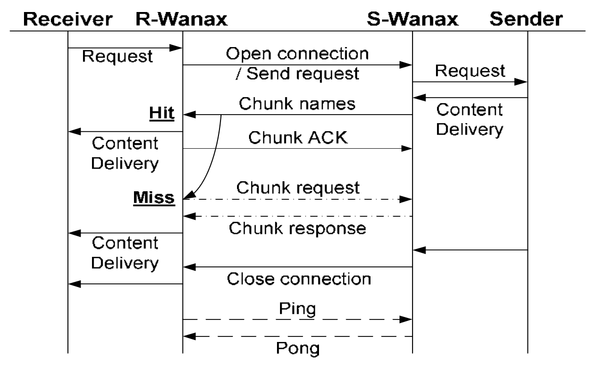


Figure 3: Basic Protocol

Wanax works by compressing redundant traffic between a pair of servers – one near the clients, called a R-Wanax, and one closer to the content, called an S-Wanax. For developing regions, the S-Wanax is likely to be placed where bandwidth is cheaper. For example, in Africa, where Internet connectivity is often backhauled to Europe via slow and expensive satellite, the S-Wanax may reside in Europe.

Since we expect most Wanax usage will be Web-related, Wanax operates on TCP streams rather than IP packets since buffering TCP flows can yield larger regions for content fingerprinting. The remote Wanax divides the incoming TCP stream into chunks and sends chunk identifiers (such as SHA-1 hashes) to the local Wanax. If the local Wanax has the chunks cached, the data is reassembled and delivered to the client. Any chunks that are not cached can be fetched from the remote Wanax or other nearby peer. Figure 2 shows the overall system architecture. Each machine is capable of acting as both S-Wanax and R-Wanax, based on the direction of communication.

3.1 Basic Protocol

Wanax uses three kinds of communication channels between the accelerators – control, data, and monitoring channels. The control channel is used for connection management and chunk name exchange. The data channels are used to request and deliver uncached chunks, so it is stateless and implemented as a simple request-reply protocol. Finally, the monitoring channel is used for checking the liveness and load levels of the peers using a simple heartbeat protocol. Figure 3 shows typical data transfer between two Wanax gateways.

Control Channel When client A initiates a TCP connection to client B in the WAN, that connection is transparently intercepted by the Wanax gateway accelerator [1], R-Wanax. R-Wanax selects S-Wanax which is network topologically closer to B, and sends it an *open connection* message with the IP and the port number of B. S-Wanax then opens a TCP connection to B and a logical end-to-end user connection between A and B is established.

[1]Non-cacheable protocols(e.g., SSH, HTTPS) are bypassed.

When the client B sends data back to S-Wanax, S-Wanax generates chunk names from the data and sends them to R-Wanax in a *chunk name* message. Each chunk name message contains a sequence number so that R-Wanax can reconstruct the original content in the right order. After R-Wanax reconstructs and delivers the chunk data to the original client, it sends a *chunk acknowledgment* (ACK) message to S-Wanax. S-Wanax can then safely discard the delivered chunks from its memory, and proceed with sending more chunk names.

When the sender or receiver closes the connection, the corresponding Wanax sends a *close connection* message to other gateway and the connections between the gateways and the clients are closed once all the data is delivered. The control channel, however, remains connected. All control messages carry flow identifiers, so one control channel can be multiplexed for many data flows. Control messages can be batched for efficiency.

Data and Monitoring Channels The data channel uses *chunk request* and *chunk response* messages to deliver the actual chunk content in case of a cache miss at R-Wanax. We also have the *chunk peek* message which is used to query if a given chunk is cached, which is used in our load shedding system.

Each Wanax accelerator monitors the status of its peers by exchanging heartbeats on the monitoring channel. The heartbeat response carries the load level of disk and network I/Os of the peer so that we can balance the request load among peers.

3.2 Multi-Resolution Chunking

MRC combines the advantages of both large and small chunks by allowing multiple chunk sizes to co-exist in the system. Wanax uses MRC to achieve (1) high compression rate, (2) low disk seeks, and (3) low memory pressure. When content overlap is high, Wanax can use larger chunks to reduce disk seeks and memory pressure. However, when larger chunks miss compression opportunities, Wanax uses smaller chunk sizes to achieve higher compression. In contrast, existing WAN accelerators typically use a fixed chunk size, which we term *single-resolution chunking*, or SRC.

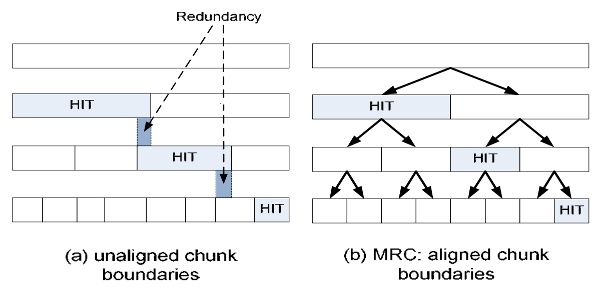

Redundancy
HIT
HIT
HIT
HIT
HIT
HIT
(a) unaligned chunk boundaries
(b) MRC: aligned chunk boundaries

Figure 4: Multi-Resolution Chunking

Scheme	Compression Rate	Disk I/O	Memory Pressure	Index Update
SRC-Small	High	High	High	Simple
SRC-Large	Low	Low	Low	Simple
MRC-Small	High	High	High	Complex
MRC-Large	High	Low	High	Complex
MRC	High	Low	Low	Simple

Table 1: Comparison of Chunking Schemes

Generating Chunks Generating multiple chunk sizes requires careful processing, not only for efficiency, but also to ensure that chunk boundaries are aligned. A naive approach to generating chunks can yield unaligned chunk boundaries, as shown in Figure 4(a). Here, the fingerprinting algorithm was run multiple times with multiple sizes. However, due to different boundary-detection mechanisms, chunk size limits, or other issues, the boundaries for larger chunks are not aligned with those of smaller chunks. As a result, when fetching chunks to reconstruct data, some areas of chunks overlap, while some chunks only partly overlap, causing wasted bandwidth when a partially-hit chunk must be fetched to satisfy a smaller missing range.

Instead, we perform a single-pass fingerprinting step, in which all of the smallest boundaries are detected, and then larger chunks are generated by matching different numbers of bits of the same boundary detection constraint. This process produces the *MRC tree* shown in Figure 4(b), where the largest chunk is the root, and all smaller chunks share boundaries with some of their leaf chunks. Performing this process using one fingerprinting pass not only produces a cleaner chunk alignment, but also requires less CPU.

Storing Chunks All chunks generated by the MRC process are stored to disk, even though the smaller chunks contain the same data as their parent. The rationale behind this decision is based on the observation that disk space is cheap, and having all chunks be fully independent simplifies the metadata [2] indexing process, reducing memory pressure in the system, also minimizing disk seeks as well. For example, when reading a chunk content from disk, MRC requires only *one* index entry access, and only *one* disk seek.

Two other options would be to reconstruct large chunks from smaller chunks, which we call *MRC-Small*, and storing the smaller chunks as offsets into the root chunk, which we call *MRC-Large*.

While both MRC-Small and MRC-Large can reduce disk space consumption by saving only unique data, they suffer from more disk seeks and higher memory pressure. To reconstruct a larger chunk, MRC-Small needs to fetch all the smaller chunks sharing the content, which can significantly increase disk access. The metadata for each small chunk is accessed in this process and loaded in memory, increasing memory pressure compared to standard MRC with only one chunk index entry. MRC-Large avoids multiple disk seeks but complicates chunk index management. When a chunk is evicted from disk or overwritten, all dependent chunks must also be invalidated. This requires either that each metadata entry grows to include all sub-chunk names, or that all sub-chunk metadata entries contain backpointers to their parents.

MRC avoids these problems by making all chunks independent of each other. This choice greatly simplifies the design at the cost of more disk space consumption. In practice, however, we can store more than one month's worth of chunk data on a single 1 TB disk assuming a 1 Mbps WAN connection. Table 1 summarizes the trade-offs of different schemes.

Content Reconstruction When an R-Wanax receives an MRC tree (chunk names only) from an S-Wanax, it builds a *candidate list* to determine which chunks can be fetched locally, at peers, and from the S-Wanax. To get this information, it queries its local cache and peers for each chunk's status, starting from the root. Since Wanax uses the in-memory index to handle this query, it does not require extra disk access. If a chunk is a hit, R-Wanax stops querying for any children of the chunk. For misses, we find the root of the subtree containing only misses, and fetch that from S-Wanax. After reconstructing the content, Wanax stores each uncached chunk in the MRC to disk for future reference.

Chunk Name Hints Optimization Sending full MRC trees would waste bandwidth if there is a cache hit at a high level in the tree or when subtrees are all cache misses. Sending one level of the tree at a time avoids the wasted bandwidth, but increases the transmission latency with a large number of round trips. Instead, we have S-Wanax predict chunk hits or misses at R-Wanax and prune the MRC tree accordingly. We augment S-Wanax with a hint table that contains recently-seen chunk names along with timestamps. Before sending the MRC tree, S-Wanax checks all chunk names against the hint table. For any hit in the hint table, S-Wanax avoids sending the subtrees below the chunk. If it is a miss or the chunk name hint is stale, S-Wanax determines the largest subtree that is a miss and sends one chunk content for the entire subtree. This way, we eliminate any inefficiency exchanging MRC trees, further increasing effective compression rate.

[2]chunk name, disk location of chunk content, and chunk length at a minimum.

Here, we assume the S-Wanax and the R-Wanax will be roughly synchronized over time – what an R-Wanax receives from an S-Wanax now is likely to be fetched from the same S-Wanax in the future. We use the timestamps to invalidate old hint entries, but even if prediction is wrong, it does not affect correctness.

3.3 Resource Sharing via Peering

Wanax incorporates a peering mechanism to share the resources such as disks, memory, and CPU with nearby peers using cheaper/faster local connectivity. It allows Wanax to distribute the chunk fetching load among the peers and utilize multiple chunk cache stores in parallel, improving performance. In comparison, existing WAN accelerators support only point-to-point communication.

To reduce scalability problems resulting from querying peers [45], Wanax uses a variant of consistent hashing called Highest Random Weight (HRW) [40]. Regardless of node churn, HRW deterministically chooses the responsible peer for a chunk. We considered other approaches like Summary cache [12], but HRW consumes small memory at the expense of more CPU cycles, and this trade-off fits well in the developing world scenario. In comparison, periodic rebuilds of a Bloom filter would require re-scanning all chunk metadata, causing significant memory pressure and possibly disk access.

Here is how it works. On receiving the *chunk name* message from S-Wanax, R-Wanax sends a *chunk request* message to its responsible peer Wanax. The message includes the missing chunk name and the address of S-Wanax from whom the name of the missing chunk originates. If the peer Wanax has the chunk, it sends the requested chunk content back to R-Wanax with a *chunk response* message. If not, the peer proxy can fetch the missing chunk from S-Wanax, deliver it to R-Wanax, and save the chunk locally for future requests. If peers are not in the same LAN and could incur separate bandwidth cost, fetching the missing chunk falls back to the R-Wanax instead of the peer. After finishing data reconstruction, R-Wanax also distributes any uncached chunk to its corresponding peers. We introduce a *chunk put* message in the data channel for this purpose.

3.4 Intelligent Load Shedding

While chunk cache hits are desirable in general since they reduce bandwidth consumption, too many disk accesses may degrade the effective bandwidth by increasing the overall latency. This problem becomes even worse in the developing world where the disk performance is poor. In such cases, we can opportunistically use network bandwidth instead of queueing more requests to the disk. By using the disk for larger chunks and fetching smaller chunks over the network, we can sustain high effective bandwidth without disk overload.

Algorithm 1 Intelligent Load Shedding

```
Require: $C$: all the chunk names to be scheduled
     $BW$, $RTT$: link bandwidth and RTT
     $Q_i$: # of pending disk requests for peer $i$
     $B_i$: pending network bytes to receive for peer $i$
     $S$: per chunk disk latency
 1: partition $C$ with HRW
 2: resolve $C$ with *chunk peek* message in parallel
 3: generate the candidate list
     $D_i$: cache-hit chunks on peer $i$
     $N$: cache-miss chunks
 4: estimate each latency
     $T_{D_i} = (|D_i| + Q_i) \times S$
     $T_N = RTT + \{\sum_i B_i + \sum_{c \in N} length(c)\}/BW$
 5: while $max(T_{D_i}) > T_N$ do
 6:     pick the peer $k$ where $max(T_{D_i}) = T_{D_k}$
 7:     move the smallest chunk from $D_k$ to $N$
 8:     update $T_{D_k}$ and $T_N$
 9: end while
10: return $D_i$ and $N$
```

We introduce intelligent load shedding (ILS), which exploits the structure of the MRC tree and dynamically schedules chunk fetches to maximize the effective bandwidth given a resource budget. The ILS algorithm is presented in Algorithm 1, and takes the link bandwidth (BW) and round-trip latency (RTT) of the R-Wanax as input. Each peer Wanax also uses the monitoring channel to send heartbeats that contain its network and disk load status in the form of the number of pending disk requests (Q_i), and the pending bytes to receive from network (B_i). We assume per-chunk disk read latency (S), or seek time is uniform for all peers for simplicity.

The first step in the ILS process is generating the candidate list. On receiving the chunk names from S-Wanax, R-Wanax runs the HRW algorithm to partition the chunk names (C) into responsible peers. Some chunk names are assigned to R-Wanax itself. Then R-Wanax checks if the chunks are cache hits by sending the *chunk peek* messages to the corresponding peers in parallel. Based on the lookup results, R-Wanax generates the candidate list (§3.2). Note that this lookup and candidate list generation process (line 2 and 3 in Algorithm 1) can be saved by name hints from S-Wanax, which R-Wanax uses to determine the results without actual lookups.

The next step in the ILS process is estimating fetch latencies for the network and disk queues. From the candidate list, we know which chunks need to be fetched over network (*network queue*, N) and which chunks need to be fetched either from local disk or a peer (*disk queues*, D_i). Based on this information, we estimate the latency for each chunk source. For each disk queue, the estimated *disk* latency will be per-chunk disk latency (S) multiplied by the number of cache hits. For the net-

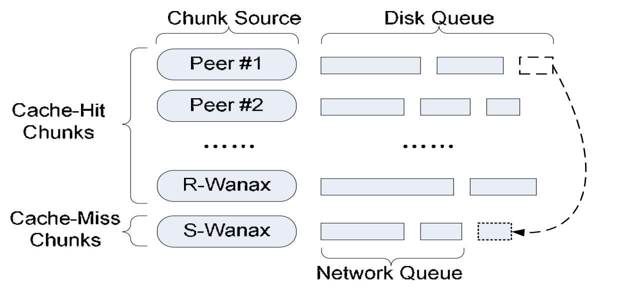


Figure 5: Intelligent Load Shedding: by moving smaller chunks from the disk queue to the network queue, the overall latency is further reduced.

work queue, the estimated *network* latency will be one RTT plus the total size of cache-miss chunks divided by BW. If there were pending chunks in the network or disk queues, each latency is accordingly adjusted. We assume the latency between the R-Wanax and peers is small, and do not incorporate it in our model.

The final step in ILS is balancing the expected queue latencies, but doing so in a bandwidth-sensitive manner. We decide if we need to move some cache hit chunks from a disk queue to a network queue – since fetching chunks from each source can be done in parallel, the total latency will be the maximum latency among them. If the network is expected to cause the highest latency, we stop here because no further productive scheduling is possible. When disk latency dominates, we can reduce it by fetching some chunks from the network. We choose the *smallest* chunk because it reduces one disk seek latency while increasing the minimum network latency. We update the estimated latencies, and repeat this process until the latencies equalize, as shown in Figure 5. After finishing ILS, R-Wanax distributes *chunk request* messages to corresponding peers. We send the requests in the order they appear in the candidate list, in order to avoid possible head-of-line (HOL) blocking.

Note that ILS algorithm works with both MRC and SRC. However, by moving the smallest chunk from the disk queue to the network queue, MRC could further reduce the disk latency than SRC, which results in smaller overall latency. Combined with MRC's better overall disk performance and compression, it gives much higher effective bandwidth.

4 Simulation Analysis

To understand the trade-offs between MRC and other schemes, we simulate their behavior under a variety of workloads, comparing bandwidth savings, disk access overheads, memory pressure, and performance.

4.1 Simulator

We develop a simulator that reads the packet-level traces from tcpdump [38] and simulates various scenarios using SRC and MRC. The simulator uses libnids [16] for stream reconstruction, and consists of 7,000 lines of C code. The outputs are actual and ideal bandwidth savings with and without chunk indexing metadata overhead, disk access overhead for chunk content fetching, and total memory usage. We use 20-byte SHA-1 hashes for the chunk names, and model point-to-point deployments with one S-Wanax and one R-Wanax with no peers. The simulator implements all of the Wanax design mentioned earlier, including the chunk name hint optimization used for both SRC and MRC.

We vary the chunk size for both schemes, with SRC using chunks from 32 bytes to 64 KB, and MRC using three tree configurations, with a 64 KB root chunk with tree degrees 2, 4 and 8 each. The child chunk size is obtained by dividing the parent chunk size by the degree. For example, a degree-2 tree ($d = 2$) starts with a 64 KB root chunk and two 32 KB children chunks. Each child chunk recursively forms a subtree with the same degree until the chunk size reaches 32 bytes. A degree-4 tree has 64 bytes as leaf node size while a degree-8 tree has 128 bytes as the minimum size. If needed, we also change the height of the MRC tree of the same degree, by controlling the smallest chunk size, m.

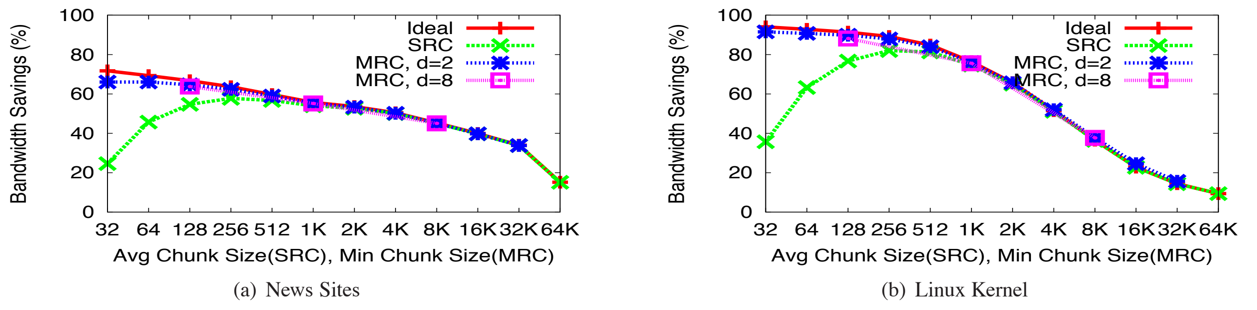
4.2 Workload

We choose two types of workloads – dynamically-generated Web content and redundant large files. We focus on dynamic content because the static content is likely to be handled by a standard Web proxy, and we can further reduce bandwidth consumption on uncacheable content with Wanax. We select a number of popular news sites, fetch the front pages every five minutes, and measure the redundancy between the fetches. [3] To generate traffic close to what actual users would produce, we use Firefox 3.0 [13] to fetch the content, and we enable the browser cache to avoid re-fetching cacheable content. We collect packet-level traces for three days, yielding a 1GB trace with 102K TCP sessions and a 72% redundancy. We refer to this workload as "news sites".

The large-file workload represents long-lived connections for videos or software packages. For this, we download two different versions of the Linux kernel source tar files, 2.6.26.4 and 2.6.26.5, one at a time and gather packet-level traces as well. The size of each tar file is about 276 MB, and the two files are 94% redundant. We refer to this workload as "Linux kernel".

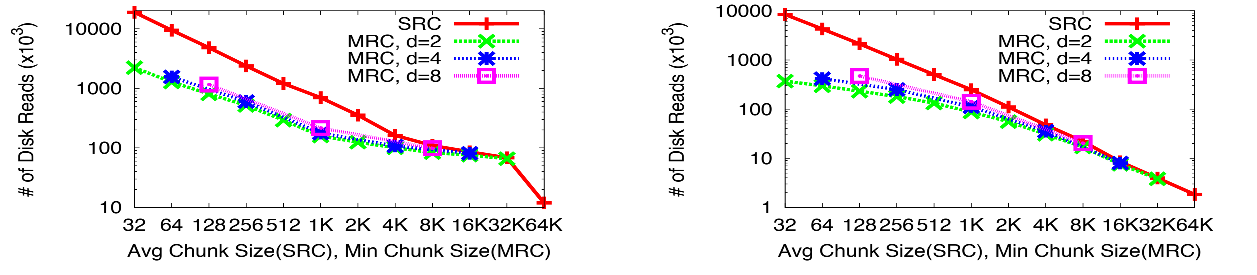
Cacheability Breakdown Table 2 separates the potential bandwidth savings on the news sites by their HTTP cacheability, as determined by checking the cache control directives in the response headers. The top two numbers represent the portion of HTTP-uncacheable bytes (H-U), while the bottom two indicate HTTP-cacheable bytes (H-C). The middle two numbers show the portion

[3] CNN, Google News, NYTimes, Slashdot, Digg, Fark, Salon, Yahoo News, and Drudgereport.



(a) News Sites (b) Linux Kernel

Figure 6: Potential Bandwidth Savings (d:degree) – SRC overheads prevent it from reaching ideal savings for smaller chunk sizes. MRC savings are close to ideal across all chunk sizes.



(a) News Sites (b) Linux Kernel

Figure 7: Disk Operation Cost (d:degree) – By using larger chunks when possible, MRC dramatically reduces the number of disk operations needed for a given workload. Note: Y axis is thousands of operations.

	SRC	MRC-2	MRC-4	MRC-8
H-U/W-U	20	20	21	23
H-U/W-C	62	62	61	59
H-C/W-C	10	10	9	8
H-C/W-U	8	8	9	10

Table 2: News Sites Cacheability Breakdown (%) – as a result of browser caching, most traffic in this workload is HTTP-uncacheable (H-U). However, it still has much redundancy, making most bytes Wanax-cacheable (W-C).

of Wanax-cacheable bytes (W-C), while the outer two depict the Wanax-uncacheable portion (W-U).

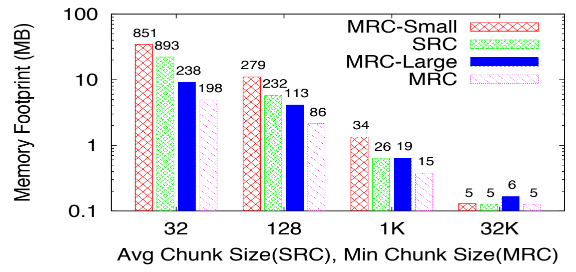
We see that most of the bytes are not cacheable by HTTP, but are cacheable by Wanax. Of the bytes that are not HTTP cacheable, about 75% are redundant and can benefit from Wanax. Of the HTTP-cacheable bytes, more than half are Wanax-cacheable as well. This result suggests that Wanax plus a browser cache can handle much of the traffic, but that Wanax with an HTTP proxy can provide even greater savings. Using an HTTP proxy with Wanax also allows HTTP-cacheable responses to be served directly from the proxy without re-contacting the content provider.

4.3 Results

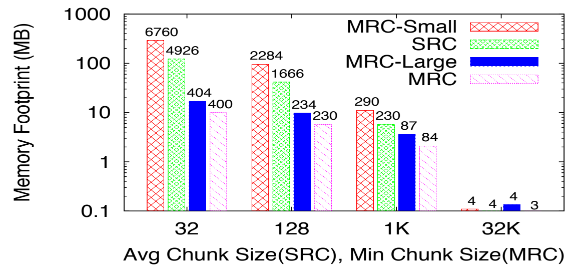
Potential Bandwidth Savings Figure 6 shows the ideal and actual bandwidth savings on both workloads for various chunk sizes. As expected, the ideal bandwidth savings increases as the chunk size decreases. However, due to the chunk indexing metadata transmission overhead, the actual savings with SRC peaks at a chunk size of 256 bytes with 58% bandwidth savings on the news sites, and 82% on the Linux kernel. The bandwidth savings drops as the chunk size further decreases, and when the chunk size is 32 bytes, the actual savings is only 25% on the news sites and 36% on the Linux kernel.

On the other hand, MRC approaches the ideal savings regardless of the minimum chunk size. With 32 byte minimum chunks, it achieves close to the maximum savings on both workloads – about 66% on the news sites and 92% on the Linux kernel. This is because MRC uses larger chunks whenever possible and the chunk name hint significantly reduces metadata transmission overheads. When comparing the best compression rates, MRC's effective bandwidth is 125% higher than SRC's on the Linux kernel while it shows 24% improvement on the news sites.

Disk Operation Cost MRC's reduced per-chunk indexing overhead becomes clearer if we look at the number of disk I/Os for each configuration, shown in Figure 7. SRC's disk fetch cost increases dramatically as the chunk size decreases, making the use of small chunks almost impossible with SRC. MRC requires far fewer disk operations even at small chunk sizes. When the leaf node chunk size is 32 bytes, SRC performs 8.5 times as many disk operations on the news sites, and 22.7 times more on the Linux kernel.

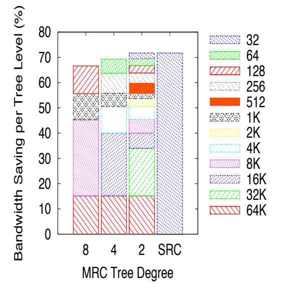


(a) News Sites

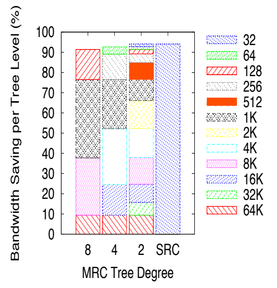


(b) Linux Kernel

Figure 8: Memory Footprint Comparison. Note log-scale Y axis. MRC's memory pressure is typically one-tenth that of SRC and MRC-Small. MRC-Large typically uses twice the memory due to backpointer overhead.



(a) News Site



(b) Linux Kernel

Figure 9: Per-level Bandwidth Savings in the MRC Tree – most MRC savings are from larger chunk sizes, reducing disk access and memory pressure.

Memory Pressure Memory pressure limits the amount of cache storage that a WAN accelerator can serve and the amount of RAM it requires for that storage. Figure 8 compares the memory footprint with different chunking approaches. We count the number of chunk index entries that are used during the simulation, and calculate the actual memory footprint. Each bar represents the memory footprint (MB), and the numbers on top of each bar show the number of used cache entries in thousands. Due to space constraints, we show only the MRC trees with the degree 2, but other results follow the same trend.

MRC incurs much less memory pressure than SRC does, since MRC requires one cache entry for any large chunk while SRC needs several cache entries for the same content. MRC-Small, however, requires even more cache entries than SRC does since reconstructing a larger chunk requires accessing all of its child entries. At a 32-byte chunk size, MRC-Small consumes almost 300 MB for the linux kernel while MRC requires only about 10 MB for the cache entries. MRC-Large shows a similar number of cache entries as MRC. However, the actual memory consumption of MRC-Large is much worse than MRC because every child chunk has a back pointer to its parent. MRC-Large consumes almost twice as much memory as MRC on the news workload.

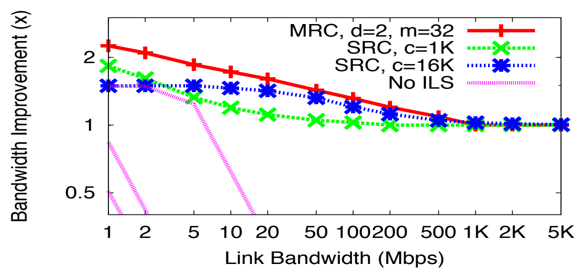
MRC Chunk Size Breakdown Figure 9 shows the breakdown of bandwidth savings by different chunk sizes. We present all three MRC configurations and SRC with a 32-byte minimum chunk. For MRC, chunk sizes are sorted from smallest at top to largest at bottom, and the bottom bar shows the root chunk size of 64KB.

The results explain MRC's low disk overhead and low memory pressure – only a small fraction of the total savings is handled by the smallest chunks with MRC, whereas all of the savings is handled by 32-byte chunks with SRC. Most of MRC's bandwidth reduction comes from larger chunks, which results in a much smaller number of disk I/Os and cache entries. We can see the similar trend across different MRC degrees. For example, the portion handled by a 4KB chunk size in MRC degree 4 is handled by 8KB chunk size as well in MRC degree 2. This means that some portion of 4KB chunks are merged into 8KB chunks in MRC degree 2. In all the MRC scenarios, chunks that are 4KB or larger provide 40-50% of the bandwidth savings, drastically reducing disk I/O.

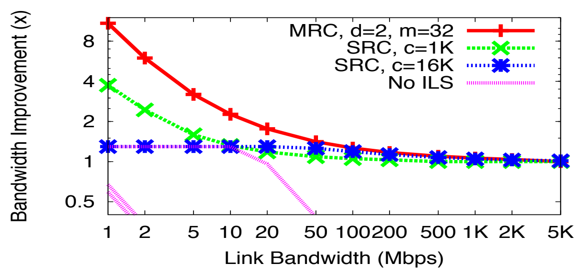
Intelligent Load Shedding Based on the previous results of bandwidth savings and disk performance, we simulate the effective bandwidth improvement (times) given a target link capacity using ILS in Figure 10. We vary the link capacity from 1Mbps to 5Gbps, and assume one 7200RPM SATA disk.

We see that the effective bandwidth improvement of both MRC and SRC approaches one as link capacity increases, but SRC drops much faster than MRC. With smaller chunk sizes, SRC shows a high effective bandwidth with slow links due to its high compression rate, but the effective bandwidth quickly degrades as the link capacity grows. This is because with small chunks, the disk soon becomes the bottleneck of the system. In the same context, SRC with larger chunk sizes performs better with fast links, but shows a worse bandwidth improvement for slow links due to its low compression rate.

MRC outperforms SRC regardless of link speed, and it sustains high effective bandwidth by leveraging multiple chunk sizes. If the link is slow, MRC fetches even the smallest chunks from disk, suppressing most redundancy. As the link capacity increases, MRC stops



(a) News Sites



(b) Linux Kernel

Figure 10: Effective Bandwidth Improvement over Link Capacity (c: avg chunk size, d: degree, m: min chunk size) – as link capacity increases and disk performance becomes a bottleneck, MRC sheds cache hits on smaller chunks first, leading to a graceful degradation in effective bandwidth. With ILS disabled, the bandwidth collapses to the bottleneck disk speed. Note log-scale Y-axis.

fetching the smaller chunks from disk, and focuses on the larger chunks rather than completely disabling compression, gracefully degrading the effective bandwidth. When ILS is disabled, the effective bandwidth of all three configurations collapses to the bottleneck disk speed.

5 Implementation

The Wanax prototype consists of about 18,000 lines of C code sharing the same MRC/SRC code base with the simulator in §4.

PPTP/GRE Tunneling To provide easy access to end users, Wanax is implemented as an Internet gateway with PPTP/GRE tunneling, with TUN/TAP [42] support planned for the near future. Currently, users need to specify the IP address of Wanax in their PPTP client on Linux (or to set up a VPN client on Microsoft Windows), after which all traffic from the user is forwarded to the Wanax system. Wanax performs content fingerprinting only on TCP streams, and bypasses all non-TCP packets.

Reconstructing TCP Byte Streams While a fully transparent solution could intercept all IP packets and reconstruct TCP streams, that creates unnecessary complexity between layer 3 and 4. Instead, we intercept each TCP connection from the client, and redirect it to Wanax. This greatly simplifies the buffering process since Wanax can use the regular socket interface to recover the original content. We implement this in the PPTP server [26] by modifying the destination address and port of the incoming packets from the client, to those of Wanax. Similar to network address translation (NAT), we store this mapping in the address translation table, and recover the original address and port for the outgoing packets from Wanax to the client. This requires about 500 lines of PPTP server code modification.

Storage System We use HashCache [6] not only as an HTTP proxy, but also as scalable storage for storing and retrieving the chunk content as well as the chunk name hint. With a highly memory-efficient indexing scheme, HashCache fully utilizes a Terabytes-sized disk with less than 256 MB of physical memory, which is an ideal storage system for developing regions. HashCache is designed to use at most one disk seek for reading a random chunk, and performs group writes of related chunks to minimize disk latency for future reading. Wanax uses two special HashCache APIs, `hc_peek()` and `hc_hint()`. `hc_peek()` tells the existence of a chunk without performing actual disk I/O, and we use it for ILS and chunk name hints. `hc_hint()` exports the queuing status of the disk I/Os and is used for ILS calculations.

Optimizing Transport Protocol Inter-Wanax communication uses a set of techniques to improve network performance over high-latency WANs. While implementing a fully-custom transport protocol might yield some additional benefit, we opt for simplicity and use TCP variants optimized for high-delay, low-bandwidth links [14, 15]. They modify the congestion avoidance algorithm so that they can quickly increase the congestion window even under high latency. In addition, Wanax multiplexes all communication over a set of long-lived TCP connections, avoiding an extra connection setup overhead of one RTT [23]. We also disable slow-start after idle time because we carefully control the number of connections per link. [4] These techniques are helpful especially for short-lived HTTP connections, which dominates traffic in the developing world [11]. In our tests, we find this combination yields close to the line speed even for many short connections.

Minimizing MRC Computation Overhead While MRC preserves high bandwidth savings without sacrificing disk performance, it consumes more CPU cycles in fingerprinting and hash calculation due to an increased number of chunks. Figure 11 shows average time for running Rabin's fingerprinting algorithm and SHA-1 on one chunk with an average size of 64 KB from a 10 MB file. Surprisingly, Rabin's fingerprinting, though it is known to be computationally efficient, turns out to be still quite expensive, taking three times more than SHA-1. How-

[4]sysctl '`tcp_slow_start_after_idle`' in Linux.

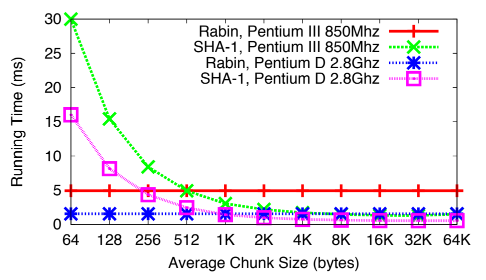


Figure 11: MRC Computation Overhead for 64KB Block

ever, the aggregate SHA-1 cost increases as MRC's leaf chunk size decreases. If naively implemented, the total CPU cost of an MRC tree with a height n would be $n \times$ Rabin's fingerprinting time + sum of SHA-1 calculation of each level.

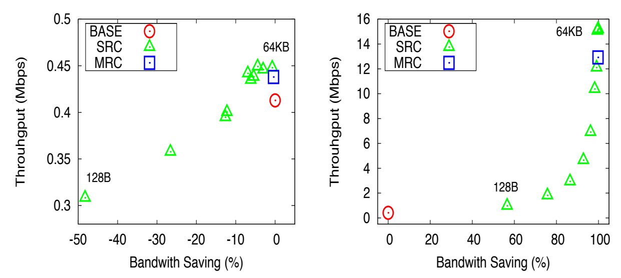
We consider two general optimizations which can be applied to both S-Wanax and R-Wanax. First, we run Rabin's fingerprinting on content only once, detect the smallest chunk boundaries, and derive the larger chunk boundaries from them. Second, we compute SHA-1 hashes only when necessary using the chunk name hint. For example, if S-Wanax knows that this chunk has been sent to R-Wanax before, S-Wanax assumes all of its children are already in R-Wanax and sends only the name of the parent. Likewise, if R-Wanax knows that a chunk has been stored on disk before, it does not re-store its children.

In addition, we implement an R-Wanax specific optimization. When the top-level chunk is a miss with R-Wanax but there are some chunk hits in the lower levels in the MRC tree, we only need to run fingerprinting with the cache-missed candidate list chunks. In order to support this, we now store a Rabin's fingerprint value (8 bytes) along with each chunk name hint. If a chunk in the candidate list is a cache hit, we can retrieve the fingerprint value for the chunk. If a chunk is a cache miss, we run the fingerprinting function to find and store any smaller chunks. We now know Rabin's fingerprint values for all chunks in the candidate list, so we can also reconstruct any parents without running the fingerprinting on the cache-hit chunks.

These optimizations are mainly for the case of chunk cache hits, where more CPU cycles are needed to deliver the chunks to the client. In case of a chunk cache miss, the bottleneck will still be in the slow WAN link for the developing worlds and consuming extra CPU cycles will not affect the download throughput.

6 Evaluation

In this section, we evaluate our prototype implementation of Wanax. Except for the realistic traffic test in the middle of this section, our tests use 1GHz AMD Athlon 64 X2 CPU machines equipped with 1GB RAM and a SATA disk. We divide them into two regions to represent the content provider and the developing region, with intra-region bandwidths set to 100Mbps. We vary the bandwidth and latency of the bottleneck WAN link connecting the two regions, depending on the evaluation scenarios. We have an origin server and an S-Wanax in the content provider side, and a client and two R-Wanax nodes in the developing region. Both the SRC and MRC tests are conducted using the same Wanax servers with the same TCP optimizations. To emulate the effect of large working sets which do not fit in memory, we disable in-memory cache for serving chunk content.



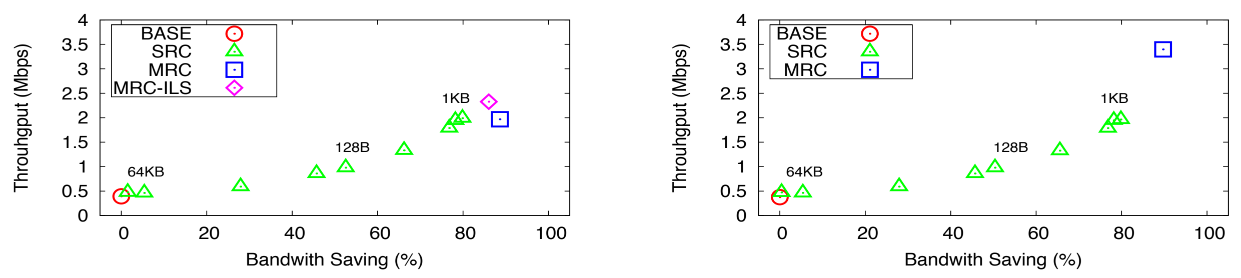
(a) 100% Cache Miss (b) 100% Cache Hit

Figure 12: Cache Miss and Cache Hit Performance – even on all-hit or all-miss workloads, the extra overheads of MRC are small compared to SRC. The best SRC performers on this set use large chunk sizes, which would produce poor compression on realistic workloads.

Microbenchmark For our microbenchmark, we use two 1 MB files that have 90% redundancy using a 64-byte chunk size. The bottleneck WAN link is set to 512Kbps with a 200ms RTT. We download the first file twice to generate a cold cache miss and a complete cache hit, and then download the second file to generate a partial cache hit. We repeat the experiment by increasing the number of peers, and performing ILS. The downloading throughput (effective bandwidth) without Wanax (BASE) is only 0.41 Mbps due to the high WAN latency. We test SRC with chunk sizes from 128 bytes to 64KB, and a degree-8 MRC using a 128-byte minimum and 64KB maximum chunks.

Figure 12 (a) shows the bandwidth savings and throughputs when downloading the first file. Since every chunk is a cache miss, S-Wanax sends the content as well as the chunk name. Due to the chunk name overhead, SRC consumes more bandwidth than BASE, with up to 48% overhead for 128-byte chunks. However, the throughput is higher than BASE, reaching 0.45Mbps for 64KB chunks, due to the optimized TCP between Wanax nodes. On the other hand, the overhead of MRC is negligible since it uses the largest chunk size of 64KB for most cache misses, yielding an overhead of 5.6% and a throughput of 0.43Mbps.

Figure 12 (b) compares MRC with SRC for a second download of the same file. As expected, SRC with the



(a) One node (b) Two nodes

Figure 13: Performance with 90% redundancy and 512Kbps WAN link – MRC without ILS produces much better compression than any SRC configuration, and throughput is comparable to the best SRC. With ILS enabled, MRC produces better compression and throughput than any SRC configuration. When peering is used, disk is not a bottleneck, and enabling ILS has no effect.

large chunk sizes (16, 32, and 64KB) shows the best throughput of 15Mbps. [5] As the chunk size decreases, the throughput degrades, and the bandwidth savings is also reduced due to the per-chunk metadata overhead. However, MRC achieves both high throughput and bandwidth savings since they use the largest chunk size in this case. The slightly lower throughput of MRC versus SRC with large chunks is because MRC generates multiple chunk sizes for the first download, spreading the layout of the large chunks on disk, whereas the SRC download stores all of the chunks in sequence on disk.

Figure 13 (a) depicts the performance of downloading the second file after warming the cache with the first file (90% redundancy). In this particular workload, SRC with 1KB chunks is the best configuration achieving both the highest bandwidth savings (80%) and highest throughput (2Mbps). MRC, in comparison, provides a higher bandwidth savings (89%) than any SRC scheme, but without ILS, the disk becomes the bottleneck and the throughput is almost the same as the best SRC. Enabling ILS raises the MRC throughput to 2.4Mbps at the cost of bandwidth savings, but beats every SRC configuration on both bandwidth savings and throughput – ILS automatically finds the sweet spot regardless of the workload.

Figure 13 (b) presents the effect of peering. The experiment is the same as the previous test, but now includes another Wanax peer in the developing region. Since peering allows Wanax to access multiple disks in parallel, we can expect improved throughputs by mitigating the disk bottleneck. However, for SRC, the lower compression rate causes the WAN bandwidth to be the bottleneck, so peering does not help. In comparison, MRC benefits significantly from peering, achieving 3.4Mbps throughput. With disk no longer the bottleneck, ILS is not necessary, and enabling it does not shed any load.

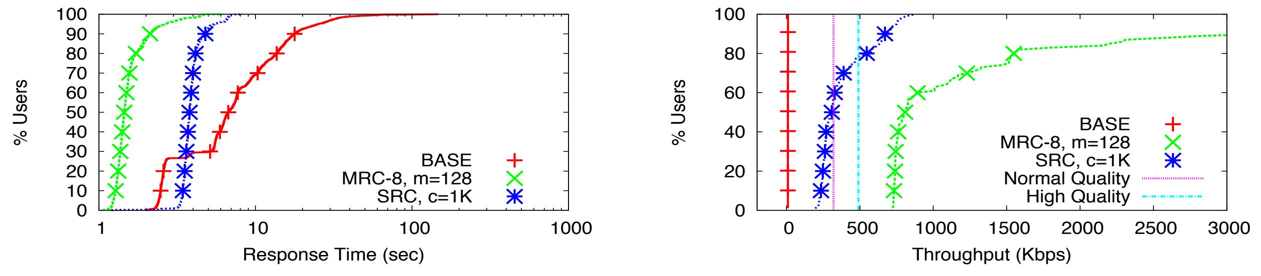
Realistic Traffic To test more general Web browsing in the developing regions, we use Alexa Top Sites [3] and YouTube [46] for testing using realistic traffic. We use the "pc850" nodes on Emulab [43], each equipped with an 850MHz Pentium III CPU and 512MB RAM. The bottleneck WAN link is set to 1Mbps with a 1000ms RTT, mimicking a satellite link commonly found in the developing world. First, we collect packet-level traces from Alexa's top 10 sites for Ghana and Nigeria, to reflect common Web browsing activity in these regions, including both cacheable and uncacheable objects. We replay 5,000 connections with 200 simultaneous clients on the traffic, and measure the response time. We also pick one of the most popular videos at the time of testing [6] from YouTube, and have 100 clients simultaneously download the whole 18 MB clip. YouTube's video content is not cacheable by standard Web proxies since its URL is in a customized format and changes for each download. This test is intended to reflect a classroom scenario where a number of students watch the same clip roughly at the same time. We introduce an 1 second interval between the client requests, and measure the throughput of each transfer. For these experiments, we use only one R-Wanax, configured with either a degree-8 MRC tree or a 1 KB SRC configuration, which has shown good performance and bandwidth savings.

Figure 14 (a) shows the response time CDFs for the Alexa workload. The average object size is 5,425 bytes and the median is 570 bytes. MRC outperforms both SRC and direct transfer (BASE), and shows the median response time of 1.5 seconds while BASE and SRC show 6.7 and 3.8 seconds each. MRC and SRC are generally faster than BASE because they fetch most objects from the local disk cache. However, on this workload, MRC typically uses one disk read per object while SRC frequently uses multiple disk I/Os per object. This behavior explains the performance difference between the two, and the disk latency sometimes makes SRC worse than BASE.

Figure 14 (b) shows the YouTube results. The bitrate of the video is 490 Kbps and the BASE curve shows

[5]The throughput is limited by the 200ms link latency since the total download time is 500-600ms. Downloading a larger file (10 MB) yields 44 Mbps throughput.

[6]The first weekly address by President Obama on 01/24/09



(a) Alexa (b) YouTube

Figure 14: Realistic Traffic – both MRC and SRC provide compression on the Alexa workload, but MRC's median response time is 1.5 seconds, compared to 3.8 for SRC. For the YouTube test, all students would be able to view the video without interruption using MRC, while with SRC, it would be 20% for the high-quality version and 50% for the low-quality version.

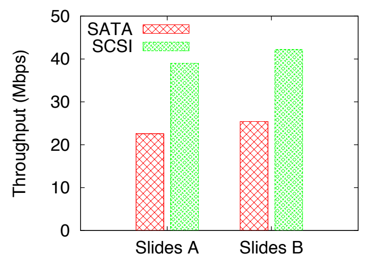


Figure 15: Enterprise Environment – with no link bottleneck, the underlying system performance can be measured. No standard test exists for these systems, but these figures are comparable to those published for commercial systems.

that nobody would be able to watch the clip reliably on a 1 Mbps link. SRC would satisfy only about 20% of the users while MRC would deliver the video to all 100 clients without interruption. The median throughputs are 809 Kbps and 309 Kbps for MRC and SRC each. We test the lower quality video (320 Kbps) of the same content, and find that SRC satisfies half of the users. Without a WAN accelerator, only two or three clients can watch the clip at any given time, which makes using classroom video problematic.

Enterprise Environment Finally, we evaluate Wanax in an enterprise-like environment to determine how well it performs compared to commercial WAN accelerators. Unfortunately, while vendors publish performance figures, none appear to publish the test scenarios they use. Testing in industry magazines uses LANs to remove network capacity as the bottleneck, which we also use in this test. That is, we focus on the impact of disk performance by separating the network delay from the overall throughput. This is because the disk performance is the bottleneck in higher link capacity enterprise environments. A high-end commercial product targeting large offices or data centers uses multiple small capacity SCSI disks, [7] rather than one large capacity disk [32].

[7] 1U product supporting 45Mbps uses 4 disks, 3U product supporting 310Mbps uses 16 disks.

We create three sets of PowerPoint slides – an original deck that is 11.9 MB, and two modified decks that add slides to this deck, yielding a 13.9 MB file (Slides A) and a 14.9 MB file (Slides B). Compared with the original deck, these have redundancies of 86% and 81%. This represents a scenario where multiple people in different offices are collaborating on a presentation. We first warm the Wanax cache with the original file, and measure the throughput of the two modified slide decks. The measured bandwidth savings correspond to the redundancy in the files. We use MRC degree 8 with the minimum chunk size of 128 bytes, and repeat the experiments with two different disks.

As shown in Figure 15, both file downloads achieve slightly more than 20 Mbps with a single 7200 RPM SATA disk at R-Wanax. The slightly larger redundancy of Slides A (86%) incurs more disk hits than Slides B (81%), and it is reflected in B's slightly larger throughput. With a faster 15K RPM SCSI disk, the throughput almost doubles in both cases. In examining the configurations of one of the leading WAN accelerator companies [32], we see that their per-disk performance ranges from 8 Mbps to 20 Mbps depending on the configuration. Since we have incomplete information about the testing scenario, we cannot draw any firm conclusions, but our range of 20-40 Mbps suggests that we have at least comparable performance to commercial solutions in these higher-end configurations, and our memory pressure analysis suggests that Wanax does so using a small fraction of the memory of these systems.

7 Related Work

Much work, both commercial and academic, has been done in the broad area of redundancy elimination for network traffic. Web caching has been an active field, with the first-generation caches [8, 17] storing unchanging objects in their entirety, often with protocol support. Later techniques included delta encoding [19] to reduce traffic for object updates, and duplicate detection to suppress downloading of aliased HTTP objects [20].

Spring and Wetherall [36] further extend the previous approaches to sub-packet granularity, and develop a protocol-independent content fingerprinting (CF) scheme that eliminates redundancy over a single link. Recently, Anand *et al.* [4] extend this idea on ISP routers, with an emphasis on redundancy-aware routing algorithms. RTS-id [2] also eliminates redundancy in the wireless environment by caching recently transferred packets through eavesdropping. However, they all work on a per-packet basis at the link layer, which limits the potential bandwidth saving to the packet size. Since Wanax operates on byte streams, it does not have such limits.

Content fingerprinting has been widely adapted in many applications, including network file systems [5, 21], Web proxies [7, 30], file transfer services [27, 28], and Web servers [24]. However, all of these systems are application-specific, and do not work across protocols. DOT [41] proposes a flexible architecture for generic data transfer, which is protocol independent, but not transparent, and requires application-level modification. Ditto [10] extends DOT, and targets wireless mesh network environments. It is complementary to Wanax since Wanax focuses on eliminating redundancy on the bottleneck WAN link.

There are a number of commercial WAN accelerators [9, 33, 35] as well. They operate below the application layer, so they are both transparent and protocol independent. However, they are designed to run on dedicated server-class appliances with fast disks and a large pool of memory. Also, their typical enterprise deployment scenario is a star topology where branch offices are speaking only to a central office. Running them on the resource-limited shared machines with mesh topology in the developing world would be problematic leading to poor performance if possible at all. Instead, Wanax is designed from the scratch to specifically address the developing world's needs, and we believe some of our techniques such as MRC and ILS can also be applied to the enterprise scenarios to reduce the deployment cost.

To the best of our knowledge, Wanax is the first system to simultaneously use multiple chunk sizes. Riverbed [33] uses a bottom-up segmentation scheme [1] that first uses 100 byte chunks, and then creates larger pseudo-chunks that contain the names of the smaller chunks [31], which is similar to MRC-Small. This approach provides some of the disk efficiency and bandwidth benefits of MRC, but still requires access to all of the metadata of the 100-byte chunks, thereby retaining the memory pressure of the smaller chunks. In the context of large file replication, Remote Differential Compression [39] uses a similar recursive segmentation scheme with a minimum chunk size of 1 KB, in order to reduce the size of chunk names sent over the network. Most recently, multi-resolution handprinting [37] proposes an efficient technique for choosing the best chunk sizes for the given similar files, by comparing handprints - a deterministic subset of chunk hashes with different chunk sizes. We share the same spirit of exploiting tradeoffs of multiple chunk sizes. However, their method is based on static analysis on the files they already have. MRC is a dynamic counterpart, and is directly applicable for online processing.

Finally, there are a number of active research projects for the developing world. DitTorrent [34] shares the same idea of exploiting better regional connectivity as Wanax, but focuses on scheduling P2P dialup connections. As systems like rural WiFi [25] or WiMAX [44] extend the Internet to new regions, Wanax can help improve the effective bandwidth delivered.

8 Conclusion

We have presented the design and implementation of Wanax, a flexible and scalable WAN accelerator targeting developing regions. Using a novel chunking technique, MRC, Wanax provides high compression and high throughput, while maintaining a small memory footprint. This profile enables it to run on resource-limited shared hardware, an important requirement in developing-world deployments. By exploiting MRC to direct load shedding, Wanax is designed to maximize the effective bandwidth even when disk performance is poor due to overloading. The peering scheme used in Wanax allows multiple servers in a region to share their resources, and thereby exploit faster and cheaper local-area connectivity instead of always using the WAN. In summary, through a careful design addressing the developing world challenges, Wanax provides customized, cost-effective WAN acceleration to the region with commodity hardware. We have begun deploying Wanax at a few partner sites in Africa, and expect to have more results about real-world operation in the future.

Acknowledgment

We would like to thank Anirudh Badam for providing HashCache, and Marc Fiuczynski for arranging and coordinating our deployments in Africa. We also thank our shepherd, Michael Isard as well as anonymous USENIX ATC reviewers. This research was partially supported by NSF awards CNS-0615237 and CNS-0916204, and KAIST awards G04100004.

References

[1] US patent #7,116,249: Content-based segmentation scheme for data compression in storage and transmission including hierarchical segment representation, 2006.

[2] Afanasyev, M., Andersen, D. G., and Snoeren, A. C. Efficiency through eavesdropping: Link-layer packet caching. In *USENIX NSDI* (Apr. 2008).

[3] ALEXA THE WEB INFORMATION COMPANY. http://www.alexa.com/.

[4] ANAND, A., GUPTA, A., AKELLA, A., SESHAN, S., AND SHENKER, S. Packet caches on routers: The implications of universal redundant traffic elimination. In *SIGCOMM* (2008).

[5] ANNAPUREDDY, S., FREEDMAN, M. J., AND MAZIERES, D. Shark: Scaling file servers via cooperative caching. In *USENIX NSDI* (2005).

[6] BADAM, A., PARK, K., PAI, V., AND PETERSON, L. Hashcache: Cache storage for the next billion. In *Proceedings of the 6th conference on Networked Systems Design and Implementation (NSDI'09)* (2009).

[7] CHAKRAVORTY, R., CLARK, A., AND PRATT, I. Optimizing web delivery over wireless links: Design, implementation and experiences. In *IEEE Journal of Selected Areas in Communications (JSAC)* (2003).

[8] CHANKHUNTHOD, A., DANZIG, P. B., NEERDAELS, C., SCHWARTZ, M. F., AND WORRELL, K. J. A hierarchical internet object cache. In *USENIX ATC* (1996), pp. 153–163.

[9] CITRIX SYSTEMS. http://www.citrix.com/.

[10] DOGAR, F., PHANISHAYEE, A., PUCHA, H., RUWASE, O., AND ANDERSEN, D. Ditto - A System for Opportunistic Caching in Multi-hop Wireless Mesh Networks. In *MobiCom* (2008).

[11] DU, B., DEMMER, M., AND BREWER, E. Analysis of WWW traffic in Cambodia and Ghana. In *WWW* (2006).

[12] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking 8*, 3 (2000), 281–293.

[13] FIREFOX WEB BROWSER. http://www.mozilla.com/firefox/.

[14] FLOYD, S. Highspeed tcp for large congestion windows. RFC 3229, 2003.

[15] HA, S., RHEE, I., AND XU, L. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Operating Systems Review. 42*, 5 (2008), 64–74.

[16] LIBNIDS. http://libnids.sourceforge.net/.

[17] MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. Performance issues of enterprise level web proxies. In *In Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1997).

[18] MANBER, U. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994).

[19] MOGUL, J., KRISHNAMURTHY, B., DOUGLIS, F., FELDMANN, A., GOLAND, Y., VAN HOFF, A., AND HELLERSTEIN, D. Delta encoding in HTTP. RFC 3229, January 2002.

[20] MOGUL, J. C., CHAN, Y. M., AND KELLY, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *USENIX NSDI* (2004).

[21] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *SOSP* (2001).

[22] ONE LAPTOP PER CHILD. http://www.laptop.org/.

[23] PADMANABHAN, V. N., AND MOGUL, J. C. Improving http latency. *Computer Networks and ISDN Systems 28*, 1-2 (1995), 25–35.

[24] PARK, K., IHM, S., BOWMAN, M., AND PAI, V. S. Supporting practical content-addressable caching with czip compression. In *USENIX ATC* (2007).

[25] PATRA, R., NEDEVSCHI, S., SURANA, S., SHETH, A., SUBRAMANIAN, L., AND BREWER, E. Wildnet: Design and implementation of high performance wifi based long distance networks. In *USENIX NSDI* (2007).

[26] POPTOP - THE PPTP SERVER FOR LINUX. http://www.poptop.org/.

[27] PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. In *USENIX NSDI* (Cambridge, MA, Apr. 2007).

[28] PUCHA, H., KAMINSKY, M., ANDERSEN, D. G., AND KOZUCH, M. A. Adaptive file transfers for diverse environments. In *USENIX ATC* (2008).

[29] RABIN, M. O. Fingerprinting by random polynomials. Tech. Rep. TR-15-81, Harvard University, 1981.

[30] RHEA, S., LIANG, K., AND BREWER, E. Value-based web caching. In *Proceedings of the Twelfth International World Wide Web Conference* (May 2003).

[31] RiOS 5.5 Technical Whitepaper. http://www.riverbed.com/docs/TechOverview-Riverbed-RiOS_5.5.pdf.

[32] RIVERBED STEELHEAD PRODUCT FAMILY DATASHEET. http://www.riverbed.com/docs/DataSheet-Riverbed-FamilyProduct.pdf.

[33] RIVERBED TECHNOLOGY, INC. http://www.riverbed.com/.

[34] SAIF, U., CHUDHARY, A. L., BUTT, S., AND BUTT, N. F. Poor man's broadband: peer-to-peer dialup networking. *SIGCOMM Computer Communication Review. 37*, 5 (2007), 5–16.

[35] SILVER PEAK SYSTEMS, INC. http://www.silver-peak.com/.

[36] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM* (2000).

[37] TANGWONGSAN, K., PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Efficient similarity estimation for systems exploiting data redundancy. In *Proc. IEEE INFOCOM* (San Diego, CA, Mar. 2010).

[38] TCPDUMP. http://www.tcpdump.org/.

[39] TEODOSIU, D., BJRNER, N., GUREVICH, Y., MANASSE, M., AND PORKKA, J. Optimizing file replication over limited-bandwidth networks using remote differential compression. Tech. Rep. MSR-TR-2006-157, Microsoft Research, Nov. 2006.

[40] THALER, D. G., AND RAVISHANKAR, C. V. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking 6*, 1 (Feb. 1998), 1–14.

[41] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for internet data transfer. In *USENIX NSDI* (2006).

[42] UNIVERSAL TUN/TAP DRIVER. http://vtun.sourceforge.net/tun/.

[43] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002).

[44] WIMAX. http://www.wimaxforum.org/home/.

[45] WOLMAN, A., VOELKER, G. M., SHARMA, N., CARDWELL, N., KARLIN, A. R., AND LEVY, H. M. On the scale and performance of cooperative web proxy caching. In *Symposium on Operating Systems Principles* (1999).

[46] YOUTUBE. http://www.youtube.com/.

An Evaluation of Per-Chip Nonuniform Frequency Scaling on Multicores *

Xiao Zhang Kai Shen Sandhya Dwarkadas Rongrong Zhong
Department of Computer Science, University of Rochester
{xiao, kshen, sandhya, rzhong}@cs.rochester.edu

Abstract

Concurrently running applications on multiprocessors may desire different CPU frequency/voltage settings in order to achieve performance, power, or thermal objectives. Today's multicores typically require that all sibling cores on a single chip run at the same frequency/voltage level while different CPU chips can have non-uniform settings. This paper targets multicore-based symmetric platforms and demonstrates the benefits of per-chip adaptive frequency scaling on multicores. Specifically, by grouping applications with similar frequency-to-performance effects, we create the opportunity for setting a chip-wide desirable frequency level. We run experiments with 12 SPECCPU2000 benchmarks and two server-style applications on a machine with two dual-core Intel "Woodcrest" processors. Results show that per-chip frequency scaling can save ∼20 watts of CPU power while maintaining performance within a specified bound of the original system.

1 Introduction and Background

Dynamic voltage and frequency scaling (DVFS) is a hardware mechanism on many processors that trades processing speed for power saving. Typically, each CPU frequency level is paired with a minimum operating voltage so that a frequency reduction lowers both power and energy consumption. Frequency scaling-based CPU power/energy saving has been studied for over a decade. Weiser *et al.* [17] first proposed adjusting the CPU speed according to its utilization. The basic principle is that when the CPU is not fully utilized, the processing capability can be lowered to improve the power efficiency. The same principle was also applied to workload concentration with partial system shutdown or dynamic DVFS in server clusters [6,7]. Bianchini and Rajamony [5] provide a thorough survey of energy-saving techniques for servers circa 2004.

When the CPU is already fully utilized, DVFS may be applied to reduce the CPU speed when running memory intensive applications. The rationale is that memory-bound applications do not have sufficient instruction-level parallelism to keep the CPU busy while waiting for memory accesses to complete, and therefore decreasing their CPU frequency will not result in a significant performance penalty. Previous studies along this direction [9, 11, 18] largely focused on exploring power saving opportunities within individual applications. Little evaluation has been done on frequency scaling for multiprogrammed workloads running on today's multicore platforms.

Multicore frequency scaling is subject to an important constraint. Since most current processors use off-chip voltage regulators (or a single on-chip regulator for all cores), they require that all sibling cores be set to the same voltage level. Therefore, a single frequency setting applies to all active cores on Intel mutlicore processors [2, 14]. AMD family 10h processors do support per-core frequency selection, but they still maintain the highest voltage level required for all cores [3], which limits power savings. Per-core on-chip voltage regulators add design complexity and die real estate cost and are a subject of ongoing architecture research [10]. Recent work by Merkel and Bellosa [13] recognized this design constraint and showed how to schedule applications on a single chip in order to achieve the best energy-delay product (EDP).

Due to the scalability limitations of today's multicores, multichip, multicore machines are commonplace. Such machines often use a symmetric multiprocessor design, with each of the multiple processor chips containing multiple cores. On these machines, nonuniform frequency scaling can still be achieved on a per-chip basis. The goal of this paper is to evaluate the potential benefits of such per-chip frequency scaling of realistic applications on today's commodity processors. To enable chip-wide frequency scaling opportunities, we group applications with similar frequency-to-performance behavior so that they run on sibling cores of the same processor chip. Using a variable-frequency performance model, we then configure an appropriate frequency setting for each chip.

Experimental Setup Our experimental platform is a 2-chip machine and each processor chip is an Intel "Wood-

*This work was supported in part by NSF grants CNS-0411127, CCF-0448413, CNS-0509270, CNS-0615045, CNS-0615139, CCF-0702505, CNS-0834451, and CCF-0937571; by NIH grants 5 R21 GM079259-02 and 1 R21 HG004648-01; and by several IBM Faculty Partnership Awards.

crest" dual-core (two cores operating at 3 GHz and sharing a 4 MB L2 cache). We modified Linux 2.6.18 to support per-chip DVFS at 2.67, 2.33, and 2 GHz on our platform. Configuring the CPU frequency on a chip requires writing to platform-specific registers, which takes around 300 cycles on our processor. Because the off-chip voltage switching regulators operate at a relatively low speed, it may require some additional delay (typically at tens of microsecond timescales [10]) for a new frequency and voltage configuration to take effect.

Our experiments employ 12 SPECCPU2000 benchmarks (applu, art, bzip, equake, gzip, mcf, mesa, mgrid, parser, swim, twolf, wupwise) and two server-style applications (TPC-H and SPECjbb2005).

2 Prototype System Design

2.1 Multichip Workload Partitioning

To maximize power savings from per-chip frequency scaling while minimizing performance loss, it is essential to group applications with similar frequency-to-performance behavior to sibling cores on a processor chip. A simple metric that indicates such behavior is the application's on-chip cache miss ratio—a higher miss ratio indicates a larger delay due to off-chip resource (typically memory) accesses that are not subject to frequency scaling-based speed reduction. We therefore group applications with similar last level cache miss ratios to run on the same multicore processor chip. We call this approach *similarity grouping*.

A natural question is how our workload partitioning would affect system performance before DVFS is applied. Merkel and Bellosa [13] profiled a set of SPEC-CPU benchmarks and found that memory bus bandwidth (rather than cache space) is the most critical resource on multicores. Based on this observation, they advocated running a mix of memory-bound and CPU-bound applications at any given time on a single multicore platform in order to achieve the best EDP. Although their *complementary mixing* appears to contradict our *similarity grouping*, in reality, they accomplish one of the same goals of more uniform memory demand. Their approach focuses on temporal scheduling of applications on a single chip, while similarity grouping focuses on spatial partitioning of applications over multiple chips. Similarity grouping has the additional advantage of being able to individually control the voltage and frequency of the separate chips.

In addition to the ability to save power by slowing down the processor without loss in performance for high miss ratio applications, miss ratio similarity grouping may lead to more efficient sharing of the cache on multicore chips. Applications typically exhibit high miss ratios because their working sets do not fit in the cache.

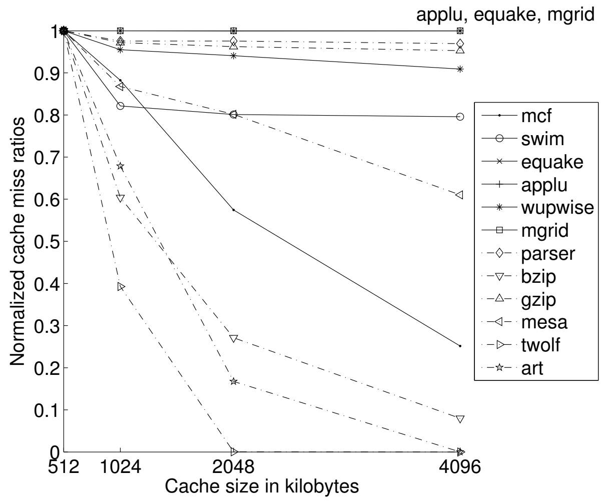


Figure 1: Normalized miss ratios of 12 SPECCPU2000 benchmarks at different cache sizes. The normalization base for each application is its miss ratio at 512 KB cache space. Cache size allocation is enforced using page coloring [20]. Solid lines mark the six applications with the highest miss ratios while dotted lines mark the six applications with the lowest miss ratios.

Increasing available cache space is not likely to improve performance significantly until the cache size exceeds the working set. This can be observed from the L2 cache miss ratio curves of 12 SPECCPU2000 benchmarks, shown in Figure 1. With the exception of mcf, most high miss ratio applications (applu, equake, mgrid, swim, and wupwise) show small or no benefits with additional cache space beyond 512 KB. In fact, the applications will aggressively occupy the cache space, resulting in adverse effects on co-running applications on sibling cores. Similarity grouping helps reduce these adverse effects by separating low miss ratio applications that may be more sensitive to cache pressure so that they run on a different chip.

2.2 Model-Driven Frequency Setting

To realize target performance or power saving objectives, we need an estimation of the target metrics at candidate CPU frequency levels. Several previous studies [9, 18] utilized offline constructed frequency selection lookup tables. Such an approach requires a large amount of offline profiling. Merkel and Bellosa employed a linear model based on memory bus utilization [13] but it only supports a single frequency adjustment level. Kotla *et al.* constructed a performance model for variable CPU frequency levels [11]. Specifically, they assume that all cache and memory stalls are not affected by the CPU frequency scaling while other delays are scaled in a linear fashion. Their model was not evaluated on real frequency scaling platforms.

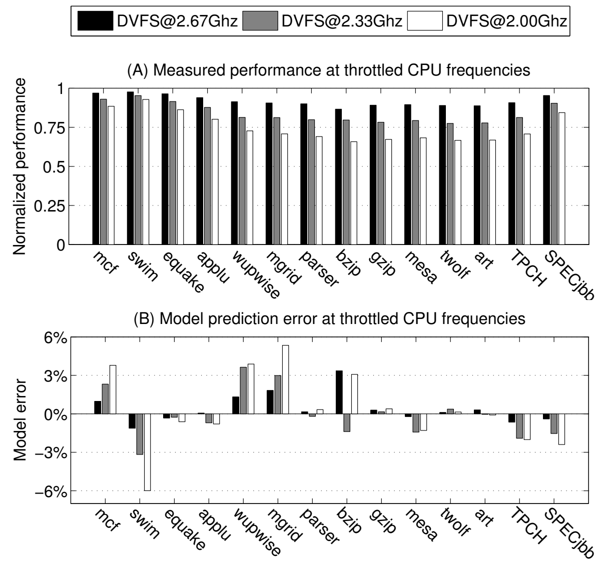


Figure 2: The accuracy of our variable-frequency performance model. Subfigure (A) shows the measured normalized performance (to that of running at the full CPU speed of 3 GHz). Subfigure (B) shows our model's prediction error (defined as $\frac{\text{prediction}-\text{measurement}}{\text{measurement}}$).

In practice, on-chip cache accesses are also affected by frequency scaling, which typically applies to the entire chip. We corrected this aspect of Kotla's model [11]. Specifically, our variable-frequency performance model assumes that the execution time is dominated by memory and cache access latencies, and that the execution of all other instructions can be overlapped with these accesses. Accesses to off-chip memory are not affected by frequency scaling while on-chip cache access latencies are linearly scaled with the CPU frequency. Let $T(f)$ be the average execution time of an application when the CPU runs at frequency f. Then:

$$T(f) \propto \frac{F}{f} \cdot (1 - R_{\text{cachemiss}}) \cdot L_{\text{cache}} + R_{\text{cachemiss}} \cdot L_{\text{memory}},$$

where F is the maximum CPU frequency. L_{cache} and L_{memory} are access latencies to the cache and memory respectively measured at full speed. We assume that these access latencies are platform-specific constants that apply to all applications. Using a micro-benchmark, we measured the average cache and memory access latencies to be around 3 and 121 nanoseconds respectively on our experimental platform. The miss ratio $R_{\text{cachemiss}}$ represents the proportion of data accesses that go to memory. Specifically, it is measured as the ratio between the L2 cache misses (L2_LINES_IN with hardware prefetch also included) and data references (L1D_ALL_REF) performance counters on our processors [8].

The normalized performance (as compared to running at the full CPU speed) at a throttled frequency f is therefore $\frac{T(F)}{T(f)}$. Since $R_{\text{cachemiss}}$ does not change across different CPU frequency settings, we can simply use the online measured cache miss ratio to determine normalized performance online. Figure 2 shows the accuracy of our model when predicting the performance of 12 SPECCPU2000 benchmarks and two server benchmarks at different frequencies. The results show that our model achieves a high prediction accuracy with no more than 6% error for the 14 applications.

The variable-frequency performance model allows us to set the per-chip CPU frequencies according to specific performance objectives. For instance, we can we can maximize power savings while bounding the slowdown of any application. The online adaptive frequency setting must react to dynamic execution behavior changes. Specifically, we monitor our model parameter $R_{\text{cachemiss}}$ and make changes to the CPU frequency setting when necessary.

Test	Chip	Similarity grouping	Complementary mixing
#1	0	{equake, swim}	{swim, parser}
	1	{parser, bzip}	{equake, bzip}
#2	0	{mcf, applu}	{mcf, art}
	1	{art, twolf}	{applu, twolf}
#3	0	{wupwise, mgrid}	{wupwise, mesa}
	1	{mesa, gzip}	{mgrid, gzip}
#4	0	{mcf, swim, equake, applu, wupwise, mgrid}	{swim, equake, applu, wupwise, gzip, twolf}
	1	{parser, bzip, gzip, mesa, twolf, art}	{mcf, mgrid, parser, bzip, mesa, art}
#5	0	2 SPECjbb threads	1 SPECjbb thread and 1 TPC-H thread
	1	2 TPC-H threads	1 SPECjbb thread and 1 TPC-H thread

Table 1: Benchmark suites and scheduling partitions of 5 tests. Complementary mixing mingles high-low miss-ratio applications such that two chips are equally pressured in memory bandwidth. Similarity grouping separates high and low miss-ratio applications on different chips (Chip-0 hosts high miss-ratio ones in these partitions).

3 Evaluation Results

3.1 Scheduling Comparison

First, we compare the overall performance of the default Linux (version 2.6.18) scheduler, complementary mixing (within each chip), and similarity grouping (across chips) scheduling policies. We design five multiprogrammed test scenarios using our suite of applications. Each test includes both memory intensive and non-intensive benchmarks. Benchmarks and scheduling partitions are detailed in Table 1.

Figure 3 compares the performance of the different scheduling policies when both chips are running at full CPU speed. For each test, the geometric mean of the applications' performance normalized to the default scheduler is reported. On average, similarity grouping is about

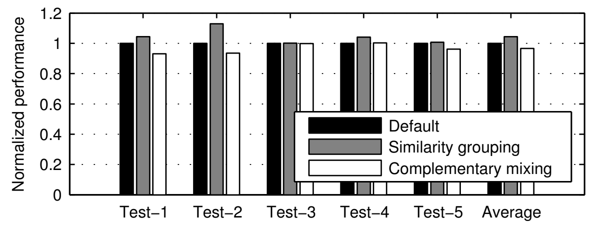


Figure 3: Performance (higher is better) of the different scheduling policies at full CPU speed.

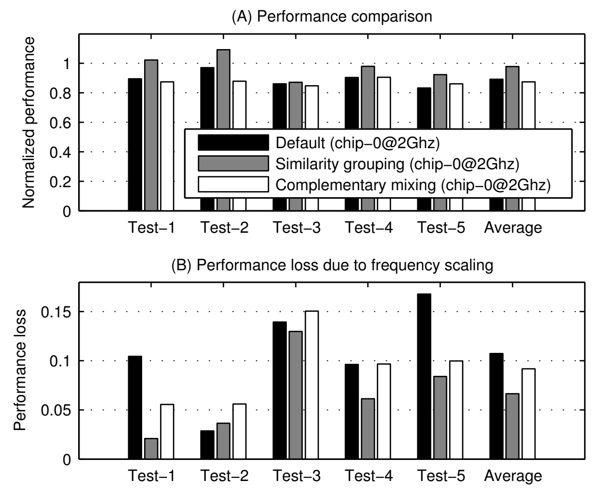


Figure 4: Performance comparisons of different scheduling policies when Chip-0 is scaled to 2 GHz. In subfigure (A), the performance normalization base is the default scheduling without frequency scaling in all cases. In subfigure (B), the performance loss is calculated relative to the same scheduling policy without frequency scaling in each case.

4% and 8% better than default and complementary mixing respectively. As explained in Section 2.1, the performance gains are due to reduced cache space interference when using similarity grouping. We also measure the power consumption of these policies using a WattsUpPro meter [1]. Our test platform consumes 224 watts when idle and 322 watts when running our highest power-consuming workload. We notice that similarity grouping consumes slightly more power, up to 3 watts as compared to the default Linux scheduler. However, the small power increase is offset by its superior performance, leading to improved power efficiency.

Next, we examine how performance degrades when the frequency of one of the two chips is scaled down. Default scheduling does not employ CPU binding and applications have equal chances of running on any chip, so deploying frequency scaling on either Chip-0 or Chip-1 has the same results. We only scale Chip-0 for similarity grouping scheduling since it hosts the high miss-ratio applications. For complementary mixing, scaling Chip-0

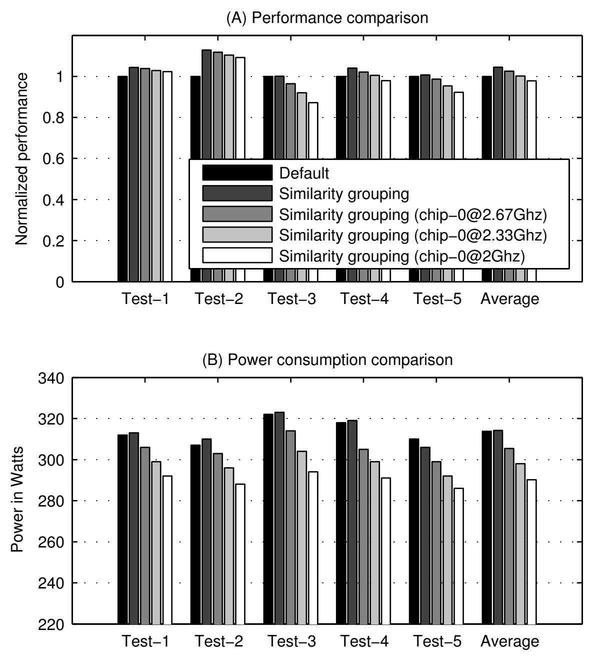


Figure 5: Performance and power consumption for per-chip frequency scaling under the similarity grouping schedule. Subfigure (B) only shows the range of active power (from idle power at around 224 watts), which is mostly consumed by the CPU and memory in our platform.

shows slightly better results than scaling Chip-1. Hence, we report results for all three scheduling policies with Chip-0 scaled to 2 GHz. Figure 4 shows that similarity grouping still achieves the best overall performance and the lowest self-relative performance loss under frequency scaling.

3.2 Nonuniform Frequency Scaling

We then evaluate the performance and power consumption of per-chip nonuniform frequency scaling under similarity grouping. We keep Chip-1 at 3 GHz and only vary the frequency on Chip-0 where high miss-ratio applications are hosted. Figure 5(B) shows significant power saving due to frequency scaling—specifically, 8.4, 15.8, and 23.6 watts power savings on average for throttling Chip-0 to 2.67, 2.33, and 2 GHz respectively. At the same time, Figure 5(A) shows that the performance when throttling Chip-0 is still comparable to that with the default scheduler.

We next evaluate the power efficiency of our system. We use *performance per watt* as our metric of power efficiency. Figure 6(A) shows that, on average, per-chip nonuniform frequency scaling achieves a modest (4–6%) increase in power efficiency over default scheduling. This is because the idle power on our platform is substantial (224 watts). Considering a hypothetical energy-

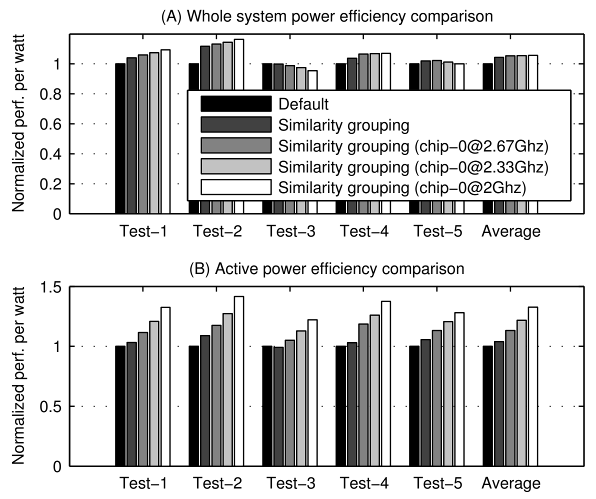


Figure 6: Power efficiency for per-chip frequency scaling under the similarity grouping schedule. Subfigure (A) uses whole system power while (B) uses active power in the efficiency calculation.

proportional computing platform [4] on which the idle power is negligible, we use the active power (full operating power minus idle power) to estimate the power efficiency improvement. In this case, Figure 6(B) shows more sizable gaps. Scaling Chip-0 at 2.67, 2.33, and 2 GHz achieves 13%, 21%, and 32% better active power efficiency respectively.

3.3 Application Fairness

While it shows encouraging overall performance, the per-chip nonuniform frequency scaling even with similarity grouping does not provide any performance guarantee for individual applications. For example, setting Chip-0 to 2 GHz causes a 26% performance loss for mgrid as compared to the same schedule without frequency scaling.

To be fair to all applications, we want to achieve power savings with bounded individual performance loss. Based on the frequency-performance model described in Section 2.2, our system will periodically (every 10 milliseconds) adjust the frequency setting if necessary to bound the performance degradation of running applications (*e.g.*, a target of 10% degradation in this experiment). Note that in this case the system may scale down any processor chip as long as the performance degradation bound is not exceeded.

Figure 7(A) shows the normalized performance of the most degraded application in each test. We observe that fairness-controlled frequency scaling is closer than the static (2 GHz) scaling to the 90% performance target. It completely satisfies the bound for three tests while it exhibits slight violations in test-3 and test-4. The most

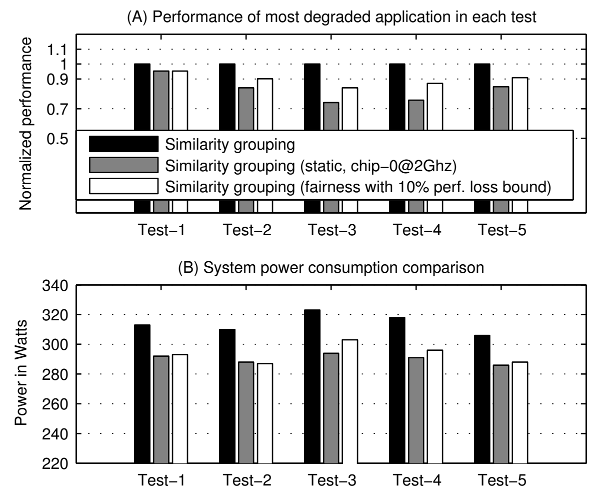


Figure 7: Performance and power consumption for static (2 GHz) and fair per-chip frequency scaling under the similarity grouping scheduling.

degraded application in these cases is mgrid, whose performance is 6% and 3% away from the 90% target in test-3 and test-4 respectively. Figure 2 shows that our model over-estimates mgrid's performance by up to 6%. This inaccuracy causes the fairness violation in test-3 and test-4. Figure 7(B) shows power savings for both static (2 GHz) and fairness-controlled frequency scaling. Fairness-controlled frequency scaling provides better quality-of-service while achieving comparable power savings to the static scheme.

4 Discussions

We have seen a slow but stable trend of increasing core numbers on a single chip, which will exacerbate the contention for memory bandwidth. Fortunately, memory technology advancement has significantly mitigated this problem. Measured using the STREAM benchmark [12], our testbed with 3 GHz CPUs (two dual-core chips) and 2GB DDR2 533 MHz memory achieves 2.6 GB/sec memory bandwidth. In comparison, a newer Intel Nehalem machine with 2.27 GHz CPUs (one quad-core chip) and 6GB DDR3 1,066 MHz memory achieves an 8.6 GB/sec memory bandwidth.

The idle power constitutes a substantial part (about 70%) of the full system power consumption on our testbed, which questions the practical benefits of optimizations on active power consumption. However, we are optimistic that future hardware designs will trend toward more energy-proportional platforms [4]. We have already observed this trend—the idle power constitutes a smaller part (about 60%) of the full power on the newer Nehalem machine. In addition, our measurement shows that per-chip nonuniform frequency scaling can reduce the average CPU temperature (by up to 5 degrees Celsius, averaged over four cores), which may lead to additional power savings on cooling.

Per-chip CPU frequency scaling is largely orthogonal to the management of shared resources on multicore processors. In particular, our frequency scaling scheme partitions applications among multiple multicore chips on the machine and mainly targets power consumption while resource management techniques such as cache space partitioning [20] and nonuniform core throttling [19] further regulate resource competition within each multicore chip.

Evaluation in this paper focuses on multiprogrammed workloads. When a single server application (consisting of many concurrent requests) runs on the machine, it may also be beneficial to group requests with similar frequency-to-performance behavior for per-chip adaptive frequency scaling. This would be possible with on-the-fly identification of request execution characteristics [15, 16] for online grouping and control.

5 Conclusion

In this paper, we advocate a simple scheduling policy that groups applications with similar cache miss ratios on the same multicore chip. On one hand, such scheduling improves the performance due to reduced cache interference. On the other hand, it facilitates per-chip frequency scaling to save CPU power and reduce heat dissipation. Guided by a variable-frequency performance model, our CPU frequency scaling can save about 20 watts of CPU power and reduce up to 5 degrees Celsius of CPU temperature on average on our multicore platform. These benefits were realized without exceeding the performance degradation bound for almost all applications. This result demonstrates the strong benefits possible from per-chip adaptive frequency scaling on multichip, multicore platforms.

References

[1] Watts Up Power Meter. https://www.wattsupmeters.com.

[2] Intel turbo boost technology in intel core microarchitecture (nehalem) based processors, Nov. 2008.

[3] AMD BIOS and kernel developer's guide (BKDG) for AMD family 10h processors, Sept. 2009.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, Dec. 2007.

[5] R. Bianchini and R. Rajamony. Power and energy management for server systems. In *IEEE Computer*, volume 37, Nov. 2004.

[6] J. Chase, D. Anderson, P. Thakar, and A. Vahdat. Managing energy and server resources in hosting centers. In *Proc. of the 18th ACM Symp. on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[7] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Proc. of the 2nd Workshop on Power-Aware Computing Systems*, Feb. 2002.

[8] IA-32 Intel architecture software developer's manual, volume 3: System programming guide, Mar. 2006.

[9] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *International Symposium on Microarchitecture*, Orlando, FL, Dec. 2006.

[10] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA'08*, Salt Lake City, UT, Feb. 2008.

[11] R. Kotla, A. Devgan, S. Ghiasi, T. Keller, and F. Rawson. Characterizing the impact of different memory-intensity levels. In *IEEE 7th Annual Workshop on Workload Characterization*, Austin, Texas, Oct. 2004.

[12] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. In *IEEE Technical Committee on Computer Architecture newsletter*, 1995.

[13] A. Merkel and F. Bellosa. Memory-aware scheduling for energy efficiency on multicore processors. In *Workshop on Power Aware Computing and Systems, HotPower'08*, San Diego, CA, Dec. 2008.

[14] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the Intel Core Duo processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[15] K. Shen. Request behavior variations. In *15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 103–116, Pittsburgh, PA, Mar. 2010.

[16] K. Shen, M. Zhong, S. Dwarkadas, C. Li, C. Stewart, and X. Zhang. Hardware counter driven on-the-fly request signatures. In *13th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 189–200, Seattle, WA, Mar. 2008.

[17] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *First USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 13–23, 1994.

[18] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Grenoble, France, Oct. 2002.

[19] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *USENIX Annual Technical Conference (USENIX)*, Santa Diego, CA, June 2009.

[20] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *4th European Conf. on Computer systems*, Nuremberg, Germany, Apr. 2009.

A DNS Reflection Method for Global Traffic Management

Cheng Huang, *Microsoft Research*
Nick Holt, *Microsoft Corporation*
Y. Angela Wang, *Polytechnic Institute of NYU*
Albert Greenberg, *Microsoft Research*
Jin Li, *Microsoft Research*
Keith W. Ross, *Polytechnic Institute of NYU*

Abstract

An edge network deployment consists of many (tens to a few hundred) *satellite* data centers. To optimize end-user perceived performance, a Global Traffic Management (GTM) solution needs to continuously monitor the performance between the users and the data centers, in order to dynamically select the "best" data center for each user. Though widely adopted in practice, GTM solutions based on active measurement techniques suffer from limited probing reachability. In this paper, we propose a novel *DNS reflection* method, which uses the GTM DNS traffic itself to measure the performance between an arbitrary end-user and the data centers. From these measurements, the best data center can be selected for the user. We have implemented and deployed a prototype system involving 17 geographically distributed locations within the Microsoft global data center network infrastructure. Our evaluation of the prototype shows that the DNS reflection method is extremely accurate and suitable for GTM. In particular, at the 95 percentile, the measured latency is 6 ms away from Ping, and the selected data center is 2 ms away from the ground-truth best.

1 Introduction

In an era where 100 ms extra delay can cost 1% drop in sales [10], cloud service providers are examining all possible measures that can reduce end-user perceived latency. One aggressive strategy is to deploy *satellite data centers* in addition to the traditional mega "backbone data centers", so as to construct an *acceleration platform* close to the end-users. Based on these satellite data centers, planet-scale *edge networks*, such as Google's CDN and Microsoft's Edge Computing Network, go beyond distributing static content and speeding up large downloads. They are increasingly important for accelerating dynamic cloud services, including search, email, maps, online office productivity software, etc.

An edge network deployment consists of many (tens to a few hundred) satellite data centers. To optimize end-user perceived performance, the "best" data center needs to be dynamically determined for each end-user. By serving users from the best data center, static content can be delivered with lower latency and higher throughput (as well as with less load on the network backbones). In addition, these satellite data centers can proxy TCP connections to speed-up dynamic cloud services [15]. One key challenge here is to find, for each end user, the best data center, which is a dynamic real-time optimization problem. In practice, the optimal selection does not always correlate well with geographic distance, but rather with a combination of network latency, packet loss, and available bandwidth. Furthermore, optimality changes as Internet routes flap, ISP relationships change, and the connectivity of physical networks fluctuates. Dynamically and accurately determining the best data center is the cornerstone of the Global Traffic Management (GTM) solution.

To optimize end-user perceived performance, the GTM solution needs to continuously monitor the performance between the users and the satellite data centers, in order to dynamically select the best data center for each user. Though widely adopted in practice, we argue that existing GTM solutions based on active measurements [1] suffer from limited probing reachability, and those based on passive measurements [4, 11, 18] incur high overhead and degrade performance. In this paper, we propose a novel *DNS reflection* method, which uses the GTM DNS traffic itself to measure the performance between an arbitrary end-user and the data centers. The contributions of this paper are as follows:

- We first survey existing DNS-based GTM solutions, including those that pick the geographically closest data center, those that use IP anycast to direct users to a data center, and those that use active probing or passive measurements. We argue that these existing solutions can perform poorly for a non-negligible fraction of the users. As part of this analysis, as a side result, we estimate that there are approximately 862,000 Local DNS (LDNS) servers used by all Windows Vista and Windows 7 users in the Internet today.

- We then propose a novel *DNS reflection* method, which uses the GTM DNS traffic itself to measure the performance between an arbitrary end-user and the data centers. The basic idea is to (very) occasionally have a user's DNS query redirected to and reflected by the DNS servers located in the satellite data centers, which can in turn measure the performance between themselves and the user. From these measurements, the best data center can be selected for the user.
- We implement and deploy a prototype system involving 17 of the geographically distributed locations in the Microsoft global data center network infrastructure. In our evaluation, we first show that the DNS reflection method is extremely accurate. In particular, at the 95 percentile, the measured latency is 6 ms away from Ping. We then compare the GTM solution based on our DNS reflection method with solutions based on geographic and anycast selection. In our experiments, the reflection-based GTM method is 2 ms within optimal at the 95 percentile, while the geography and anycast based GTM solutions are 74 ms and 183 ms from optimal, respectively. In other words, for the users whose performance is most precarious, the benefit of reflection-based GTM is significant.

2 Brief Overview of GTM Solutions

GTM is often implemented through a DNS system. As a simple example, suppose `CloudService.com` has an infrastructure of mega and satellite data centers. When a user wants to connect to the service, it first performs a DNS resolution for `CloudService.com`. The authoritative DNS server for `CloudService.com` responds with the IP address of the "best" data center, which has been determined from its GTM system. The GTM system provides, via the authoritative DNS server, different satellite data centers for different users.

Before presenting our approach to GTM, in this section we briefly review various GTM solutions and discuss related work.

2.1 Geography-based GTM Solutions

This type of GTM system uses geographic locations to map clients to data centers [7, 9]. Using commercial GeoLocation databases provided by Akamai, Quova, MaxMind and so on, each client's IP address is mapped to a geographic location. The data center chosen for a client is simply the data center that is geographically closest. Such a solution can work reasonably well for a large fraction of the clients, as recently shown in evaluation [3]. However, geographic-based solutions are still subject to well-known issue of Triangular Inequality Violation (TIV) of Internet distances. Moreover, a geographic-based solution ignores the dynamic nature of the Internet, such as the variation of latency and packet loss, and always assigns the same data center to a particular client.

2.2 Anycast-based GTM Solutions

This type of GTM uses IP anycast [14], for which all the data centers announce the same anycast IP address. When a client sends a packet to the anycast address, the packet is routed to the *anycast-closest* data center. The anycast-closest is governed by both intra- and inter-domain routing algorithms and policies. Although the anycast-closest data center is often the best data center in terms of latency for many clients, there are a non-negligible percentage of violations [5, 8]. In addition, anycast-based GTM solutions ignore packet loss, which could severely impact many delay sensitive online services.

2.3 GTM Solutions based on Active Measurements

Commercial GTM solutions commonly rely on active probing techniques to measure the performance between data centers and clients. For instance, the F5 3-DNS system actively probes Local DNS (LDNS) servers and uses the response time to calculate the round trip time and packet loss between the LDNS and the data center [1]. Observations collected at Internet honey-pots also suggest commercial CDNs, such as Akamai, are conducting large scale Internet measurements [13]. However, as we will soon demonstrate, active probing suffers from limited reachability, as many LDNSes are configured to never respond to active probes. Because a large percentage of the LDNSes cannot be probed, the effectiveness of active measurements is limited.

2.4 GTM Solutions based on Passive Measurements

An alternative to active probing is to infer performance between clients and data centers through passive monitoring. For instance, latency can be calculated by examining the gap between SYN-ACK and client ACK during the TCP three-way handshake. In order to monitor the performance between clients and every data center, such solutions require redirecting clients to sub-optimal data centers from time to time [4, 11, 18]. Although only a small number of clients will be selected to probe remote data centers, these "unfortunate" clients could suffer significant performance degradation. Because even a simple response to a web search query can take 4-6 TCP rounds trips, the inflated RTT to a remote data center can significantly degrade the user's perceived performance. Furthermore, for large responses, such as online maps or documents, directing clients to remote data centers could further inflate the response time. In an era where half a second latency kills user satisfaction [10], such degradation can become unacceptable.

Moreover, in order to minimize the impact of suboptimal redirection to clients arriving subsequently, a small (or even 0) TTL should be set in the DNS response for the initial client. Unfortunately, as Pang et al. [2] discovered in a large-scale DNS study, a significant fraction of clients and LDNSes do *not* adhere to DNS TTLs. Responses could be used long after their expiration, even in excess of 2 hours. In those cases, suboptimal redirection can degrade the performance of a large number of subsequent clients.

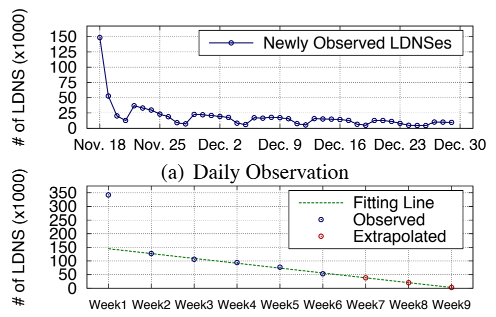
2.5 Other Factors

Most end-hosts are not far away from their LDNSes. Our latest evaluation of more than 1.6 million end-hosts accessing a popular Microsoft online service shows that the geographic distance between end-host and LDNS is less than 27 km in 60% of the cases and less than 428 km in 80% of the cases [6]. Still, there are nontrivial amount of end-hosts using LDNSes not in nearby locations, as similarly reported in earlier studies [12, 17]. The client-LDNS *mismatching* problem is being addressed independently [6]. In this paper, we focus on how to achieve good performance for those end-hosts that co-locate with their LDNSes.

Besides performance, there are many additional important factors to a GTM solution. The dynamic load on the data centers is one such factor: clients should not be directed to over-loaded data centers. ISP delivery cost is another factor. The data centers can use different ISPs, which may have different cost structures, due to complicated contractual relationships between ISPs and data center operators [8]. Taking into account delivery cost could bring significant savings to service providers, operational cost of data centers could also be explored. For instance, the power costs of the data centers can be explored so as to achieve additional savings [16]. Nevertheless, all these cost concerns are secondary and they can only be explored when they do not lead to performance degradation.

3 Limitation of Active Measurements

In this section, we set out to answer the following questions by examining a large LDNS population: how many LDNSes can be reached by active probing?



(a) Daily Observation

(b) Weekly Observation

Figure 1: Newly Observed LDNSes

3.1 A Large LDNS Population

Network Connectivity Status Indicator (NCSI) is a service running on Windows Vista or Windows 7 machines to detect the status of Internet connectivity. For instance, it shows as a system tray icon to notify users upon loss of Internet connectivity. Part of the NCSI service performs DNS queries for a special host name – `www.msftncsi.com`.

Between Nov. 18th and Dec. 30th 2009, we have sniffed 5% of the DNS traffic on the authoritative server of `msftncsi.com` for 6 weeks. A large collection of LDNS addresses is obtained. In particular, the NCSI collection contains about 795,000 LDNS addresses, located in 10,012 cities over 229 countries[1]. Figure 1(a) plots the number of uniquely newly-observed LDNSes every day. It is clear that a large number of LDNSes are observed in the first few days. However, new LDNSes keep being discovered over the entire course. A weekly pattern is also observed where the troughs correlate nicely with weekends.

To estimate the total LDNS population, Figure 1(b) plots the number of uniquely observed LDNSes every week. Except for the first week, there appears to be a clear linear trend. After simply curve fitting and extrapolation, we estimate the total number of LDNS to be around 862,000. Given the wide deployment of Windows machines, we expect this counts for a significant portion of the entire LDNS population on the Internet.

3.2 Reachability of Active Probing

We can use active probing to measure the performance between a LDNS and a data center. In this section, we study how many LDNSes can be reached via active probes.

To this end, we randomly select 50,000 LDNS addresses from the NCSI collection. Our evaluation shows that 24,660 LDNSes respond to Ping – about 49%. For

[1]from Akamai's GeoLocation database.

the rest, since they are DNS servers in nature, we issue DNS queries against them as a measure of active probing. Latency can be obtained by simply calculating the time difference between issuing a request and receiving the response. We experimented with three types of queries: 1) resolving DOT (the root DNS name); 2) reversely resolving localhost (i.e., 127.0.0.1); and 3) reversely resolving the LDNS' own IP address. Unfortunately, only 2896 (about 6% of the total) LDNSes respond to our DNS probes. Thus far, it is clear that a large percentage (about 45%) of the LDNSes are *closed* – they do *not* respond to either Ping or DNS queries from random clients.

To address the insufficiency of active measurements, in the next section, we proposed a much more involved passive DNS reflection method, which works for all LDNSes.

4 The DNS Reflection Method

4.1 The Key Idea

DNS reflection uses DNS traffic itself to measure the performance between a LDNS and a target data center. Very occasionally, the GTM DNS queries are redirected to the target data center. Since DNS traffic is UDP-based, getting one DNS query from the LDNS does *not* allow the target to infer the performance. Therefore, the target *reflects* the DNS query and responds in such a way that the LDNS is triggered to immediately issue another DNS query against the target. By examining the time difference between the two queries occurred on the target, the performance from the LDNS can be readily inferred.

DNS reflection is a passive measurement method and in this sense, similar to the approaches taken by [4, 11, 18], which redirect HTTP traffic from clients to the target. For all these methods, when there is no traffic from the clients or the LDNSes, there is no redirection and thus no measurement.

Beyond this similarity, however, the DNS reflection method differs fundamentally from [4, 11, 18] in a number of important ways: 1) DNS reflection only redirects DNS traffic, not HTTP traffic. Hence, the clients will always be served by the "best" data center (per the choice of the GTM). Although there is a latency incurred when a LDNS is elected to probe a remote date center, there is no latency inflation for subsequent HTTP transactions. 2) Each DNS reflection incurs two round trips between the LDNS and the target. This is a much smaller penalty, compared to redirecting HTTP transactions where the response time will be 4-6 times (or even more) of the inflated round trip time. 3) In HTTP traffic redirection, due to LDNS caching, subsequent arriving clients can be affected. While in DNS reflection, since the final DNS resolution result is *not* modified, it will *not* affect subsequent clients, which will always be served by the "best" data center.

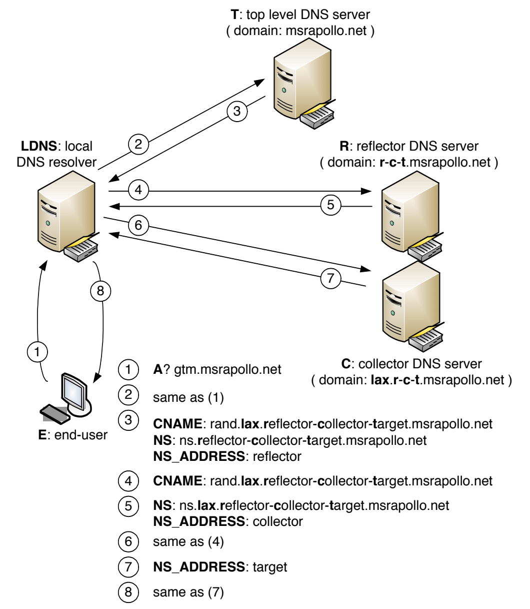

T: top level DNS server
(domain: msrapollo.net)
LDNS: local DNS resolver
R: reflector DNS server
(domain: **r**-**c**-**t**.msrapollo.net)
C: collector DNS server
(domain: **lax**.**r**-**c**-**t**.msrapollo.net)
E: end-user
1 2 3 4 5 6 7 8
1 **A**? gtm.msrapollo.net
2 same as (1)
3 **CNAME**: rand.**lax**.**r**eflector-**c**ollector-**t**arget.msrapollo.net
NS: ns.**r**eflector-**c**ollector-**t**arget.msrapollo.net
NS_ADDRESS: reflector
4 **CNAME**: rand.**lax**.**r**eflector-**c**ollector-**t**arget.msrapollo.net
5 **NS**: ns.**lax**.**r**eflector-**c**ollector-**t**arget.msrapollo.net
NS_ADDRESS: collector
6 same as (4)
7 **NS_ADDRESS**: target
8 same as (7)

Figure 2: The DNS Reflection Method

4.2 Detailed Process

Figure 2 illustrates the details of each step of the passive DNS reflection method, as elaborated in the following:

Step 1 and 2: An end-user submits a DNS query for `gtm.msrapollo.net` to its LDNS, which then forwards the query to the top level authoritative name server of `msrapollo.net`.

Step 3: Instead of responding with a target IP address, the top level domain server decides to delegate the DNS resolution to a sub-domain, whose server locates in a target data center (e.g., the one in LAX). To this end, it constructs a CNAME (an alias in DNS parlance, which itself has to be recursively resolved by DNS), which embeds LAX as the target, as well as the IP addresses of two DNS servers in LAX, denoted as *reflector* and *collector*, respectively. In addition, it delegates the CNAME to be handled by a sub-domain server, by appending the sub-domain server name and its IP address (the address of the reflector) in the DNS response.[2]

Step 4: The LDNS follows the delegation by the top level authoritative name server and sends the query of the CNAME to the reflector.

Step 5: The reflector further delegates the DNS resolution to another sub-domain, which is a sub-domain of the previously delegated sub-domain. Similarly, the reflector appends the new sub-domain server name and its IP address (the address of the collector) in the DNS response.

[2]CNAMEs and name server records should always be unique such that the entire reflection process avoids caching at LDNS completely to ensure deterministic estimation.

Step 6: The LDNS continues to follow the delegation by the reflector and send the query of the CNAME to the collector.

Step 7 and 8: The collector examines the CNAME and responds with the address of target, which is embedded in the CNAME. The DNS resolution completes.

When the reflector and the collector are in the same physical location (LAX here), we can simply calculate the network latency between the LDNS and the LAX data center from the time difference between the reflector and the collector receiving request (4) and (6), respectively. The process can be further simplified by configuring one physical machine in LAX to own both IP addresses of the reflector and the collector.

Note that all the information regarding how to respond to a particular DNS query is embedded in the query itself. Therefore, neither the top level authoritative name server, nor the reflector or the collector, needs to maintain status at any step during the reflection process. This is an important design to simplify system implementation.

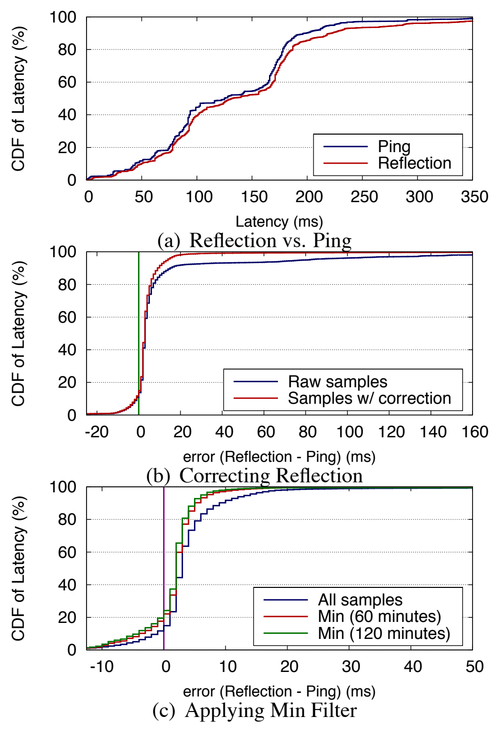
5 Evaluation

We implement a prototype system using C#, which consists of two types of DNS servers. The first type is a top level authoritative name server that responds to a GTM probing query (such as `gtm.msrapollo.net`) with a CNAME, following **step 3** in the previous section. The second type combines the reflector and the collector together and responds to queries targeting at either. It is deployed on a single physical machine configured with two IP addresses (one for the reflector and the other for the collector), in each of the 17 geographically distributed locations in the Microsoft global data center network (3 in Asia, 6 in Europe, 7 in US and 1 in Australia).

5.1 How Accurate is DNS Reflection?

In this section, we first evaluate whether the DNS reflection method gives correct latency measurement. After all, if a LDNS does *not* behave as we understand it would, or if it does *not* immediately send a second query after receiving the delegation response from a reflector, the reflection method could result in inflated or even completely wrong estimates.

To evaluate the correctness and accuracy of DNS reflection, we use 274 PlanetLab nodes as clients to issue DNS queries to our system. Every 15 minutes, each client generates 17 queries, which are redirected to all the 17 reflection locations, respectively. Upon receiving a DNS query, the reflector/collector also probes whether the requesting LDNS responds to Ping, and if it does, 6 Ping probes are sent to the LDNS. The experiment lasted 4 days during the first week of Jan., 2010. Among the 274 PlanetLab nodes, 240 of them are in the same location as their LDNSes (from Akamai's GeoLocation database). Among those co-located LDNSes, 162 of them respond to Ping. For comparison purpose, in the rest of the section, we focus on these 162 LDNSes.



(a) Reflection vs. Ping
(b) Correcting Reflection
(c) Applying Min Filter

Figure 3: Latency Comparison

For each reflection measurement, we compute the latency as outlined in the previous section. Also, we use the minimum of the 6 Ping probes as the ground-truth RTT. To compare the DNS reflection method and Ping, it is sufficient to use all the measurements collected from any single data center. Figure 3(a) shows the cumulative distributions of the two methods from one selected date center.

At the first sight, it appears that the two CDFs match each other quite well. However, if we calculate the difference between corresponding measurement samples and plot the distribution, as shown by the "Raw Samples" curve in Figure 3(b), it becomes clear that the latency measured by reflection and Ping do *not* really match well – the difference is 80 ms at the 95 percentile.

Manual examination of the samples reveals that whenever there is a large gap between reflection and Ping, the reflection latency is always twice as that of Ping. This triggered us to examine the logs of the top level authoritative name server. Finally, we discovered that some LDNSes do *not* use the delegated name server address returned by the reflector in **step 5**. Instead, it always resolves the name server address from the top level authoritative DNS server. This involves an extra round trip between the LDNS and the top level DNS server. In this particular case, the top level DNS server happens to be in the same data center, which is why the reflection latency is twice as that of Ping. Among the 162 LDNSes, there are 27 behaving this way. After we correct the samples from these LDNSes by halving the latency values, the curve "Samples w/ correction" in Figure 3(b) shows that the different with Ping is extremely small – 14 ms at the 95 percentile.[3]

The difference between reflection and Ping is even smaller if we apply an minimum filter on the measurement samples. As shown in Figure 3(c), if we take the minimum value of all the samples in a 2-hour window, then the difference with Ping is only 6 ms at the 95 percentile. Therefore, we conclude that DNS reflection is an extremely accurate measurement method.

5.2 How Good is a Reflection-based GTM?

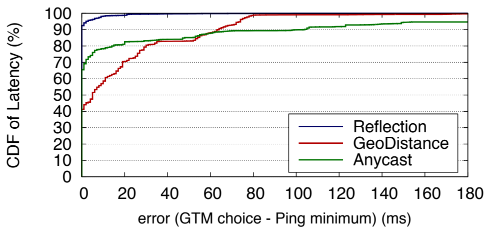


Figure 4: GTM Policy Comparison

Next, we evaluate the effectiveness of a reflection-based GTM by comparing it with geography-based GTM and IP anycast-based GTM. For the geography-based GTM, we use Akamai's GeoLocation database to find the latitude and longitude of each LDNS. The data center with the shortest great circle distance from the LDNS is selected as the best choice. For the IP anycast-based GTM, we setup an IP anycast address, which is announced from all the 17 locations. There are 17 DNS servers listening on the anycast address (one in each location). The PlanetLab nodes send DNS queries towards the anycast address. These queries are naturally routed to the anycast-closest data center. For the reflection-based GTM, we use the reflection measurements collected in every 2 hours to rank all data centers with respect to each LDNS. The minimum latency one is chosen as the best choice for the next 2 hours.

For each GTM, because of the co-location of the LDNS and its corresponding PlanetLab node, we use the Ping latency between the LDNS and the GTM-choice as the latency between the node and the data center, We use the minimum Ping latency of the 17 RTTs to all the data centers as the optimal latency. We calculate the difference between each GTM and the optimal. The cumulative distributions of the three GTMs are shown in Figure 4. We observe that the reflection-based GTM is 2 ms within the optimal (at the the 95 percentile), while the geography-based GTM is 74 ms and the anycast-based GTM is 183 ms. In other words, for the users whose performance is most precarious, the benefit of reflection-based GTM is significant!

[3]We use DNS fingerprinting tool `fpdns` to find out the DNS server software and versions, but the extra name server resolution appears to happen independently. At the moment, we can only conjecture that it is likely due to specific configurations.

Fortunately, the extra resolution incurs fixed latency and thus will *not* affect the relative performance ranking with respect to all the locations. In addition, the latency can be reduced by deploying the top level authoritative DNS server in every data center and on IP anycast.

6 Conclusions

In this paper, we argue that existing GTM solutions can perform poorly for a non-negligible fraction of the users. We propose a novel DNS reflection method, which uses the GTM DNS traffic itself to measure the performance between an arbitrary end-user and a data centers, with extremely good accuracy. We show that reflection-based GTM is very close to optimal and can significantly benefit a non-negligible fraction of the users.

References

[1] 3-DNS reference guide. WhitePaper, F5 Networks, Inc., 2002.

[2] ADITYA, J. ET AL. On the Responsiveness of DNS-based Network Control. In *Proc. of IMC* (2004).

[3] AGARWAL, S. ET AL. Match-Making for Online Games and Other Latency-Sensitive P2P Systems. In *Proc. of SIGCOMM* (2009).

[4] ANDREWS, M. ET AL. Clustering and Server Selection Using Passive Monitoring. In *Proc. of INFOCOM* (2002).

[5] BALLANI ET AL. A Measurement-based Deployment Proposal for IP Anycast. In *Proc. of IMC* (2006).

[6] HUANG, C. ET AL. Public DNS Systems and Global Traffic Management. *Work in progress* (2010).

[7] GWERTZMAN, J. ET AL. The Case for Geographical Push-Caching. In *Proc. of HotOS* (1995).

[8] HUANG, C. ET AL. Measuring and Evaluating Large-Scale CDNs. In *Microsoft Research Technical Report MSR-TR-2008-106* (2008).

[9] KARGER, D. ET AL. Web Caching with Consistent Hashing. In *Proc. of WWW* (1999).

[10] KOHAVI, R. ET AL. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers not to the Hippo. In *Proc. of KDD* (2007).

[11] KRISHNAN, R. ET AL. Moving Beyond End-to-End Path Information to Optimize CDN Performance. In *Proc. of IMC* (2009).

[12] MAO, Z. M. ET AL. A Precise and Efficient Evaluation of the Proximity between Web Clients and their Local DNS Servers. In *Proc. of USENIX ATC* (2002).

[13] OBERHEIDE, J. ET AL. Characterizing Dark DNS Behavior. In *Proc. of DIMVA* (2007).

[14] PARTRIDGE, C. ET AL. RFC1546: Host Anycasting Service, 1993.

[15] PATHAK, A. ET AL. Measuring and Evaluating TCP Splitting for Cloud Services. In *Proc. of PAM* (2010).

[16] QURESHI, A. ET AL. Cutting the Electric Bill for Internet-Scale Systems. In *Proc. of ACM SIGCOMM* (2009).

[17] SHAIKH, A. ET AL. On the Effectiveness of DNS-based Server Selection. In *Proc. of INFOCOM* (2001).

[18] STEMM, M. ET AL. A Network Measurement Architecture for Adaptive Applications. In *Proc. of INFOCOM* (2000).

An Analysis of Power Consumption in a Smartphone

Aaron Carroll
NICTA and University of New South Wales
Aaron.Carroll@nicta.com.au

Gernot Heiser
NICTA, University of New South Wales and Open Kernel Labs
gernot@nicta.com.au

Abstract

Mobile consumer-electronics devices, especially phones, are powered from batteries which are limited in size and therefore capacity. This implies that managing energy well is paramount in such devices.

Good energy management requires a good understanding of where and how the energy is used. To this end we present a detailed analysis of the power consumption of a recent mobile phone, the Openmoko Neo Freerunner. We measure not only overall system power, but the exact breakdown of power consumption by the device's main hardware components. We present this power breakdown for micro-benchmarks as well as for a number of realistic usage scenarios. These results are validated by overall power measurements of two other devices: the HTC Dream and Google Nexus One.

We develop a power model of the Freerunner device and analyse the energy usage and battery lifetime under a number of usage patterns. We discuss the significance of the power drawn by various components, and identify the most promising areas to focus on for further improvements of power management. We also analyse the energy impact of dynamic voltage and frequency scaling of the device's application processor.

1 Introduction

Mobile devices derive the energy required for their operation from batteries. In the case of many consumer-electronics devices, especially mobile phones, battery capacity is severely restricted due to constraints on size and weight of the device. This implies that energy efficiency of these devices is very important to their usability. Hence, optimal management of power consumption of these devices is critical.

At the same time, device functionality is increasing rapidly. Modern high-end mobile phones combine the functionality of a pocket-sized communication device with PC-like capabilities, resulting in what are generally referred to as *smartphones* [11]. These integrate such diverse functionality as voice communication, audio and video playback, web browsing, short-message and email communication, media downloads, gaming and more. The rich functionality increases the pressure on battery lifetime, and deepens the need for effective energy management.

A core requirement of effective and efficient management of energy is a good understanding of *where* and *how* the energy is used: how much of the system's energy is consumed by which parts of the system and under what circumstances.

In this paper we attempt to answer this question and thus provide a basis for understanding and managing mobile-device energy consumption. Our approach is to measure the power consumption of a modern mobile device, the Openmoko Neo Freerunner mobile phone, broken down to the device's major subsystems, under a wide range of realistic usage scenarios.

Specifically, we produce a breakdown of power distribution to CPU, memory, touchscreen, graphics hardware, audio, storage, and various networking interfaces. We derive an overall energy model of the device as a function of the main usage scenarios. This should provide a good basis for focusing future energy-management research for mobile devices.

Furthermore, we validate the results with two additional mobile devices at a less detailed level: the HTC Dream and Google Nexus One. Along with the Freerunner, these three devices represent approximately the last three to four years of mobile phone technology.

The paper is structured as follows. In Section 2 we describe our measurement platform and benchmarking methodology. Section 3 describes each experiment and presents the results, and in Section 4 we perform a coarse-grained validation of the results. We then analyse this data in Section 5. Section 6 surveys existing work. Finally, we conclude in Section 7.

2 Methodology

Our approach to profiling energy consumption is to take physical power measurements at the component level on a piece of real hardware. In this section, we describe the hardware and software used in the experiments, and explain our benchmarking methodology.

There are three elements to the experimental setup: the device-under-test (DuT), a hardware data acquisition (DAQ) system, and a host computer.

2.1 Device under test

The DuT was the Openmoko Neo Freerunner (revision A6) mobile phone. It is a 2.5G smartphone featuring a large, high-resolution touchscreen display, and many of the peripherals typical of modern devices. Table 1 lists its key components. The notable differences between our device and a modern smartphone are the lack of a camera and 3G modem.

Component	Specification
SoC	Samsung S3C2442
CPU	ARM 920T @ 400 MHz
RAM	128 MiB SDRAM
Flash	256 MiB NAND
Cellular radio	TI Calypso GSM+GPRS
GPS	u-blox ANTARIS 4
Graphics	Smedia Glamo 3362
LCD	Topploy 480×640
SD Card	SanDisk 2 GB
Bluetooth	Delta DFBM-CS320
WiFi	Accton 3236AQ
Audio codec	Wolfson WM8753
Audio amplifier	National Semiconductor LM4853
Power controller	NXP PCF50633
Battery	1200 mAh, 3.7 V Li-Ion

Table 1: Freerunner hardware specifications.

This device was selected because the design files, particularly the circuit schematics [7], are freely available. This is critical for our approach to power measurement, which relies on understanding the power distribution network at the circuit level. For this reason, few other devices would be suitable.

The high-level architecture of the Freerunner is shown in Figure 1. The total system memory is split equally between two banks, one external RAM package, and one on-chip. All peripherals except the graphics chip communicate with the application processor (CPU) by programmed I/O over various serial buses.

The other devices studied, the HTC Dream (G1) and Google Nexus One (N1), are described in Section 4.

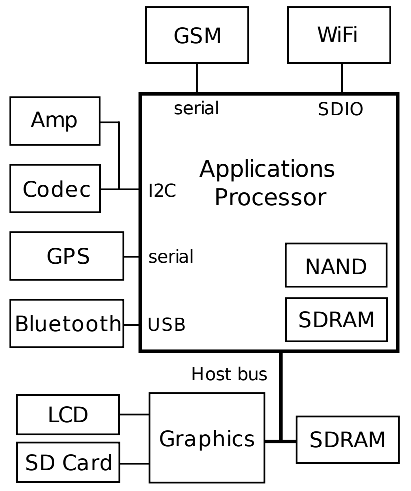


Figure 1: Architecture of the Freerunner device, showing the important components and their interconnects.

2.2 Experimental setup

To calculate the power consumed by any component, both the supply voltage and current must be determined.

To measure current, we inserted sense resistors on the power supply rails of the relevant components—this is relatively simple on the DuT selected, since most of them have been designed with placeholders for sense resistors, factory-populated with 0 Ω. Where this was not the case, choke inductors could be reused in the same way. In both cases, we replaced the part with a current-sense resistor selected such that the peak voltage drop did not exceed 10 mV, which in all cases is less than 1 % of the supply voltage and therefore presents an acceptably small perturbation. With a known resistance and measured voltage drop, current can be determined by Ohm's law.

To measure the voltages, we used a National Instruments PCI-6229 DAQ, to which the sense resistors were connected via twisted-pair wiring. The key characteristics of this hardware are summarised in Table 2.

Characteristic	Value
Max. sample rate	250 kS/s
Input ranges	±0.2 V, ±1 V, ±5 V and ±10 V
Resolution	16 b
Accuracy	112 μV @ ±0.2 V range
	1.62 mV @ ±5 V range
Sensitivity	5.2 μV @ ±0.2 V range
	48.8 μV @ ±5 V range
Input impedance	10 GΩ

Table 2: National Instruments PCI-6229 DAQ specifications [6].

The sense-resistor voltage drops were sampled differentially at the ±0.2 V input range. We used the same physical connections to measure supply voltages; these were taken relative to ground from the component side of the resistors, in the ±5 V range.

We were able to directly measure the power consumed by the following components: CPU core, RAM (both banks), GSM, GPS, Bluetooth, LCD panel and touchscreen, LCD backlight, WiFi, audio (codec and amplifier), internal NAND flash, and SD card. Since the graphics module had too many supply rails to measure directly, we instead used a combination of direct and subtractive measurements.

Power to the DuT was supplied through a bench power supply connected to the phone's battery terminals so we did not need to deal with battery management. This also prevents the OS's power policies from interfering with the benchmarks. Total system power consumption was measured at this point by inserting a sense resistor between the supply and the phone. For the G1 and N1 we measured total system power by inserting a sense resistor between the device and its battery.

Measuring backlight power required special attention, because its supply voltage (10–15 V, depending on the brightness) far exceeded the maximum range supported by our DAQ hardware. To resolve this, we pre-scaled the backlight voltage with some external circuitry, consisting of a high-input-impedance voltage follower feeding a fixed voltage divider. This brought the voltage within the ±5 V range.

2.2.1 Voltage regulation efficiency

Our measurement approach yields the power directly consumed by each component. However, a certain amount of additional power is lost in converting the supply (i.e. battery) voltage to the levels required by the components. We have not included this factor in the results reported, because the conversion efficiencies are unknown. However, based on the data sheet of a similar part (the NXP PCF 50606), the efficiency conversion is likely to be in the range of 75–85 %, depending on the current drawn.

Because of this, we differentiate between "total power", measured at the battery, and "aggregate power", measured as the sum of individual component measurements. The latter assumes no power is consumed in the non-instrumented components, and while we haven't been able to measure precisely what their contribution is, it is certainly less than 10 %, and probably within a few percent of the aggregate consumption.

One exception to this is the backlight boost converter, the efficiency of which we measured to be 67 %. We determined the cause of this poor efficiency to be heating in an external component. We found no evidence to suggest this is an issue for any of the other voltage regulators.

2.3 Software

The DuT ran the Freerunner port of the Android 1.5 operating system [1] using the Linux v2.6.29 kernel. Except for the CPU micro-benchmark, the kernel was configured with the `ondemand` frequency scaling governor, using 100 MHz and 400 MHz—the only two frequencies supported by both the hardware and OS.

On the host system we ran the power-data collection software which interfaced with the National Instruments DAQmxBase 3.3 library to collect raw data from the DAQ, aggregate it, and write the result to file for post-processing. Each data point collected was an average of 2000 consecutive voltage samples. We configured the tool such that a complete power snapshot of the system could be generated approximately every 400 ms.

The benchmarks were coordinated on the host machine, which communicated with the DuT via a serial connection. It was responsible for executing benchmarks on the DuT, synchronising the power measurement software with the benchmark, and collecting other relevant data.

2.4 Benchmarks

We ran two types of benchmarks. First, a series of micro-benchmarks designed to independently characterise components of the system, particularly their peak and idle power consumption.

Second, we ran a series of macro-benchmarks based on real usage scenarios. For low-interactivity applications (e.g. music playback), we simply launched them from the command line. For interactive applications, such as web browsing, we took a trace-based approach. A trace consisted of a sequence of input events, including a time-stamp, the name of the device providing the input (the touchscreen or one of two push-buttons), and for touchscreen events, the coordinates of the touch. The Linux kernel provides this information by reading from the `/dev/input/event*` device files. To collect the trace, we used the target application normally, while in the background storing the input events to file. We then replayed the events under benchmarking conditions by writing the collected data to the `/dev/input/event*` files at the correct time.

Although this approach does bypass the hardware and interrupt paths that would usually be followed for a touchscreen event, our measurements showed the additional power to be negligible. The vast majority of energy required to handle a touchscreen event is consumed in delivering it from the kernel to software.

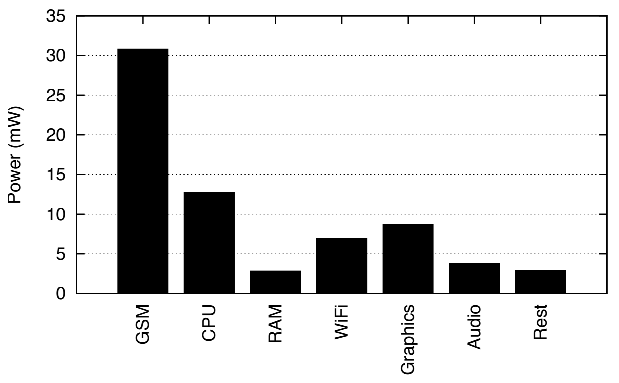


Figure 2: Power breakdown in the suspended state. The aggregate power consumed is 68.6 mW.

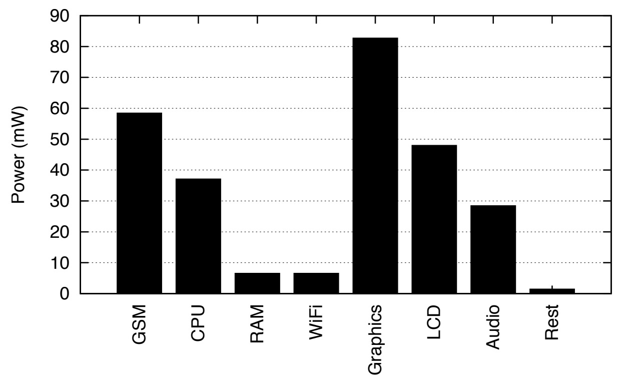


Figure 3: Average power consumption while in the idle state with backlight off. Aggregate power is 268.8 mW.

3 Results

3.1 Baseline cases

Prior to running any benchmarks, we established the baseline power state of the device, when no applications are running. There are two different cases to consider: *suspended* and *idle*. For the idle case, there is also the application-independent power consumption of the backlight to consider.

3.1.1 Suspended device

A mobile phone will typically spend a large amount of time in a state where it is not actively used. This means that the application processor is idle, while the communications processor performs a low level of activity, as it must remain connected to the network be able to receive calls, SMS messages, etc. As this state tends to dominate the time during which the phone is switched on, the power consumed in this state is critical to battery lifetime.

The Android OS running on the application processor aggressively suspends to RAM during idle periods, whereby all necessary state is written to RAM and the devices are put into low-power sleep modes (where appropriate). To quantify power use while suspended, we forced the device into Android's suspended state and measured the power over a 120 second period. Figure 2 shows the results, averaged over 10 iterations. The average aggregate power is 68.6 mW, with a relative standard deviation (RSD) of 8.2 %. The large fluctuations are largely due to the GSM (14.4 % RSD) and graphics (13.0 %) subsystems.

The GSM subsystem power clearly dominates while suspended, consuming approximately 45 % of the overall power. Despite maintaining full state, RAM consumes negligible power—less than 3 mW. Note that the GSM subsystem in our device does not use system memory—it has its own bank of RAM which we include in the GSM power measurements.

3.1.2 Idle device

The device is in the idle state if it is fully awake (not suspended) but no applications are active. This case constitutes the static contribution to power of an active system. We run this case with the backlight turned off, but the rest of the display subsystem enabled.

Figure 3 shows the power consumed in the idle state. As with the suspend benchmark, we ran 10 iterations, each of 120 seconds in the idle state. Power consumed in this state was very stable, with an RSD of 2.6 %, influenced largely by GSM, which varied with an RSD of 30 %. All other components showed an RSD below 1 %.

Figure 3 shows that the display-related subsystems consume the largest proportion of power in the idle state—approximately 50 % due to the graphics chip and LCD alone, and up to 80 % with backlight at peak brightness. GSM is also a large consumer, at 22 % of aggregate power.

3.1.3 Display

Figure 4 shows the power consumed by the display backlight over the range of available brightness levels. That level is an integer value between 1 and 255, programmed into the power-management module, used to control backlight current. Android's brightness-control user-interface provides linear control of this value between 30 and 255.

The minimum backlight power is approximately 7.8 mW, the maximum 414 mW, and a centred slider corresponds to a brightness level of 143, consuming 75 mW. The backlight consumes negligible power when disabled (as in the above idle benchmarks).

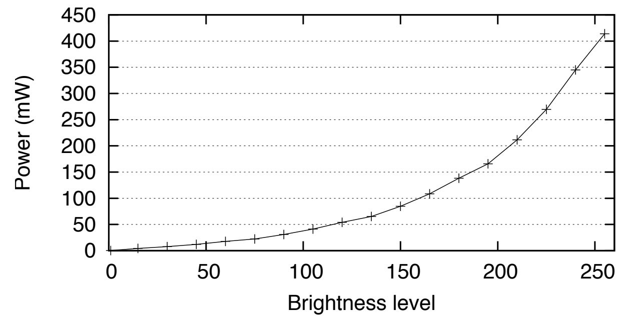


Figure 4: Display backlight power for varying brightness levels.

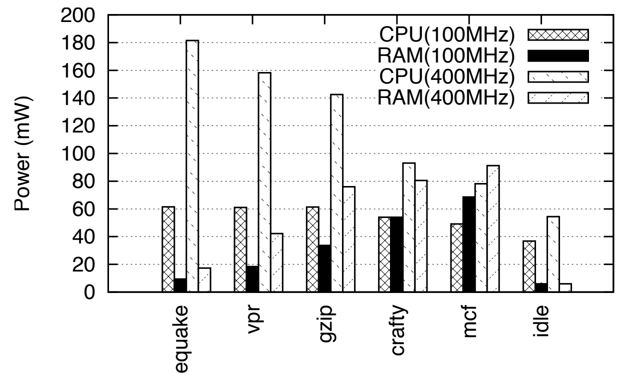


Figure 5: CPU and RAM power when running SPEC CPU2000 micro-benchmarks, sorted by CPU power.

We also measured how the content displayed on the LCD affected its power consumption: 33.1 mW for a completely white screen, and 74.2 mW for a a black screen. Display content can therefore affect overall power consumption by up to 43 mW.

3.2 Micro-benchmarks

As mentioned in Section 2.4, we used micro-benchmarks to determine the contribution to overall power from various system components. Specifically we used benchmarks to exercise the application processor (CPU and memory), the flash storage devices, and the network interfaces.

3.2.1 CPU and RAM

To measure CPU and RAM power, we ran a subset of the SPEC CPU2000 suite. There are several reasons for not running all benchmarks of the suite. Firstly, we could only use benchmarks which we could build and run on the Android OS, which rules out those written in C++ or Fortran, due to Android's lack of run-time support for these languages. They also needed to fit into the phone's limited memory and their execution times needed to be short enough to give reasonable turn-around. Finally, we were only interested in establishing the power consumption of CPU and memory, rather than making comparisons between different platforms' algorithms, hence completeness of the suite was not a relevant consideration.

From the candidates remaining according to the above criteria, we selected a set representing a good spectrum of CPU and memory utilisation, from highly CPU-bound to highly memory-bound. We determined memory-boundedness by running the entire suite on a server Linux system and comparing the slowdown due to frequency scaling. Snowdon et al. [9] show that this slowdown is primarily due to memory-boundedness. While we do not expect the benchmarks to behave similarly on the different platforms, our aim is only to select benchmarks with different characteristics.

The SPEC CPU2000 benchmarks ultimately selected are `equake`, `vpr`, `gzip`, `crafty` and `mcf`.

For each of the benchmarks, we measured the average CPU and RAM power at fixed core frequencies of 100 MHz and 400 MHz. We also measured power for the system in the idle state. Figure 5 shows these results, averaged over 10 runs. The RSD is less than 3 % in all cases.

For the idle, `equake`, `vpr` and `gzip` workloads, CPU power dominates RAM power considerably at both frequencies. However, `crafty` and `mcf` show that RAM power can exceed CPU power, albeit by a small margin.

Table 3 shows the effect of frequency scaling on the performance, as well as combined CPU and RAM power and energy of the benchmarks. The wide range of slowdown factors across the different benchmarks validates our selection of workloads as representing a range of CPU/memory utilisations.

Benchmark	Performance	Power	Energy
`equake`	26 %	36 %	135 %
`vpr`	31 %	40 %	125 %
`gzip`	38 %	43 %	112 %
`crafty`	63 %	62 %	100 %
`mcf`	74 %	69 %	93 %
idle	-	71 %	-

Table 3: SPEC CPU2000 performance, power and energy of 100 MHz relative to 400 MHz. Both CPU and RAM power/energy are included.

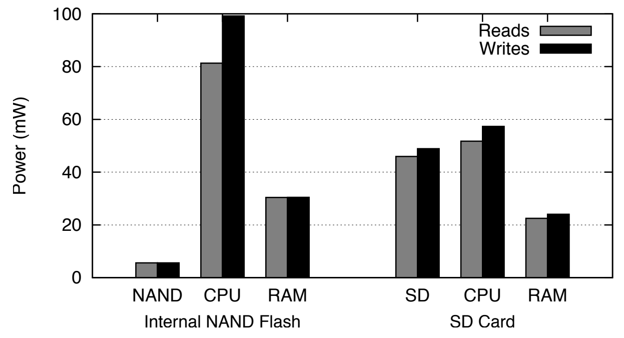


Figure 6: SD, NAND, CPU and RAM power for flash storage read and write benchmarks.

3.2.2 Flash storage

Bulk storage on the Freerunner device is provided by 256 MiB of internal NAND flash, and an external micro Secure Digital (SD) card slot. To measure their maximum power consumption, we used the Linux `dd` program to perform streaming reads and writes. For reads we copied a 64 MiB file, filled with random data, to `/dev/null` in 4 KiB blocks. For writes, 8 MiB of random data was written, with an `fsync` between successive 4 KiB blocks to ensure predictability of writes. Between each iteration we forced a flush of the page cache.

Figure 6 shows the power consumed by the NAND flash and SD card, as well as the CPU and RAM, averaged over 10 iterations of each workload. Table 4 shows the corresponding data throughput, efficiency (including NAND/SD power and the CPU and RAM power to support it), and idle power consumption. The power and throughput RSD is less than 5 % in all cases.

The graphics module, which contains the physical SD card interface, showed a power increase of 2.2 mW (2.6 % above static) for writes, and a 21.1 mW increase (26 %) for reads.

Metric	NAND	SD
Idle (mW)	0.4	1.4
Read		
throughput (MiB/s)	4.85	2.36
efficiency (MiB/J)	65.0	31.0
Write		
throughput (KiB/s)	927.1	298.1
efficiency (MiB/J)	10.0	5.2

Table 4: Flash storage power and performance.

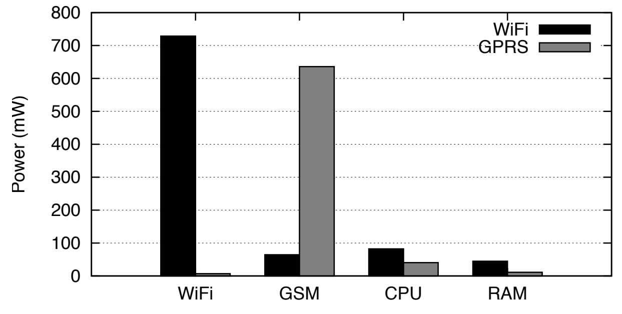


Figure 7: Power consumption of WiFi and GSM modems, CPU, and RAM for the network micro-benchmark.

3.2.3 Network

In this benchmark we stressed the two main networking components of the device: WiFi and GPRS (provided by the GSM subsystem). The test consisted of downloading a file via HTTP using `wget`. The files contained random data, and were 15 MiB for WiFi, and 50 KiB for GPRS. The results of 10 iterations of the benchmark are shown in Figure 7.

WiFi showed a throughput of 660.1 ± 36.8 KiB/s, and GPRS 3.8 ± 1.0 KiB/s. However, they both show comparable power consumption far exceeding the contribution of the RAM and CPU. The increased CPU and RAM power for WiFi reflects the cost of processing data with a higher throughput. Despite highly-variable throughput, GSM showed a relatively consistent power consumption with an RSD of approximately 2 %.

To test the effect of signal strength on power and throughput, we re-ran the network benchmarks with the device shielded within a metal box of 2 mm thickness. Over GPRS, this resulted in an increase of GSM power of 30 %, but no effect on throughput. The shielding resulted in a reported signal strength drop of 10 dBm. Over WiFi, the signal strength dropped by only 2 dBm, and no effect on throughput or power consumption was observed.

3.2.4 GPS

To measure power consumption of the GPS subsystem, we enabled the module and ran the `GPS Status 2` Android application. Table 5 shows the power consumed by the GPS module in three situations; using only the internal antenna, with an external active antenna attached, and when idle (i.e. powered down).

We noticed that the energy consumption of the module is largely independent of the received signal—neither the number of satellites, nor the signal strength, had any appreciable effect.

This observation is contrary to the part's data sheet [10], which specifies that power consumption should drop by approximately 30 % after satellite acquisition. It is unclear why we did not see such behaviour; perhaps due to the GPS module itself, or more likely an error in hardware integration or software. In addition, the power-management features of the device are not exploited by software. Thus, these figures should only be considered worst-case.

State	Power (mW)
Enabled (internal antenna)	$143.1 \pm 0.05\,\%$
Enabled (external antenna)	$166.1 \pm 0.04\,\%$
Disabled	0.0

Table 5: GPS energy consumption.

3.3 Usage scenarios

Here we show the results of using macro-benchmarks to determine power consumption under a number of typical usage scenarios of a smartphone. Specifically we examined audio and video playback, text messaging, voice calls, emailing and web browsing.

3.3.1 Audio playback

This benchmark is designed to measure power in a system being used as a portable media player. The sample music is a 12.3 MiB, 537-second stereo 44.1 kHz MP3, with the output to a pair of stereo headphones. The measurements are taken with the backlight off (which is representative of the typical case of someone listening to music or podcasts while carrying the phone in their pocket). However, GSM power was included, as the realistic usage scenario includes the phone being ready to receive calls or text messages.

Figure 8 shows the power breakdown for this benchmark at maximum volume, averaged over 10 iterations. The audio file is stored on the SD card. Between successive iterations we forced a flush of the buffer cache to ensure that the audio file was re-read each time.

The results show the audio subsystem (amplifier and codec) consuming 33.1 mW with an RSD of less than 0.2 %. Approximately 58 % of this power is consumed by the codec, with the remaining 42 % used by the amplifier. Compared with the idle state, this corresponds to a negligible change in codec power, with amplifier power increasing by 80 %. Overall, the audio subsystem accounts for less than 12 % of power consumed.

In addition to maximum volume, we also measured the system at 13 % volume. This showed little change—the audio subsystem power decreased by 4.3 mW (approximately 14 %), mostly in the amplifier. However, for unknown reasons, the power consumed by the graphics chip increased by 4.6 mW. As a result, the additional power consumed in the high-volume benchmark is less than 1 mW compared with the low-volume case.

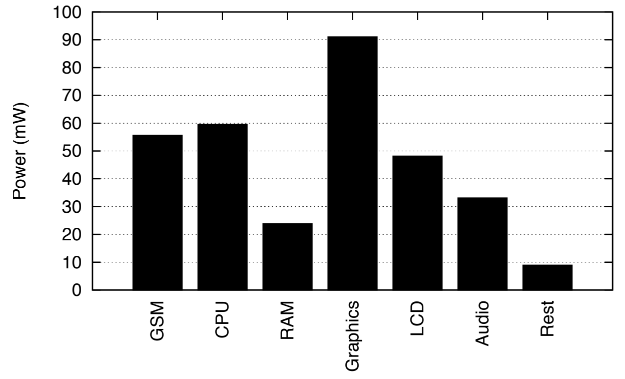


Figure 8: Audio playback power breakdown. Aggregate power consumed is 320.0 mW.

Again, maintaining a connection to the GSM network requires a significant and highly variable amount of power, specifically 55.6 ± 19.7 mW in this case. While the MP3 file is loaded from the SD card, the cost of doing so is negligible at $< 2\,\%$ of total power.

3.3.2 Video playback

In this benchmark we measured the power requirements for playing a video file. We used a 5 minute, 12.3 MiB H.263-encoded video clip (no sound), and played it with Android's camera application. Again we forced a flush of the buffer cache between iterations. The power averaged over 10 iterations is shown in Figure 9.

Since the purpose of the macro-benchmarks is to analyse the full system, we have included backlight power in the results. However, rather than arbitrarily choosing a single brightness, we have plotted the results at 0 %, 33 %, 66 %, and 100 %, corresponding to the position of Android's brightness-control slider. These correspond to brightness levels of 30, 105, 180 and 255 respectively. GSM power is again included.

While the CPU is the biggest single consumer of power (other than backlight), the display subsystems still account for at least 38 % of aggregate power, up to 68 % with maximum backlight brightness. The energy cost of loading the video from the SD card is negligible, with an average power of 2.6 mW over the length of the benchmark.

3.3.3 Text messaging

We benchmarked the cost of sending an SMS by using a trace of real phone usage. This consists of loading

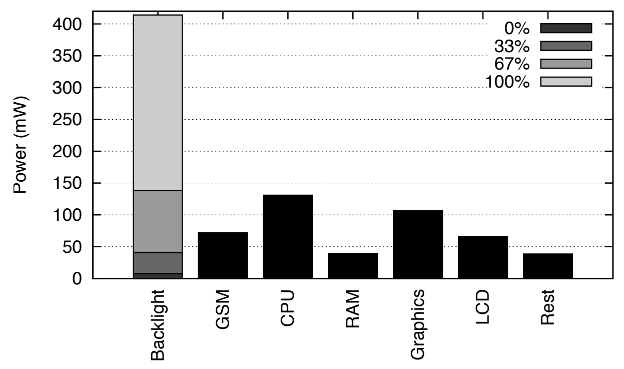


Figure 9: Video playback power breakdown. Aggregate power excluding backlight is 453.5 mW.

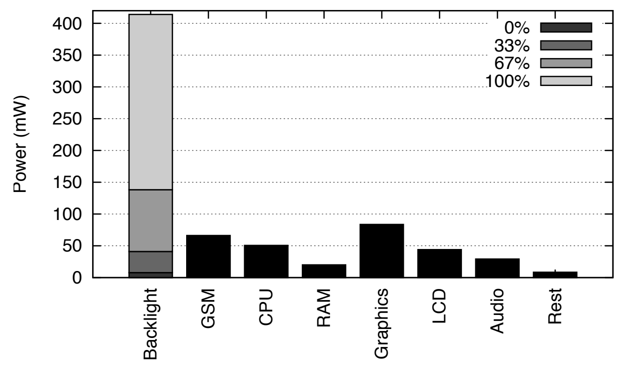


Figure 10: Power breakdown for sending an SMS. Aggregate power consumed is 302.2 mW, excluding backlight.

the contacts application and selecting a contact, typing and sending a 55-character message, then returning to the home screen; lasting a total of 62 seconds. To ensure the full cost of the GSM transaction is included, we measured power for an additional 20 seconds. The average result of 10 iterations of this benchmark are shown in Figure 10. Again, the power for four backlight brightness levels is shown.

Power consumed is again dominated by the display components. The GSM radio shows an average power of 66.3 ± 20.9 mW, only 7.9 mW greater than idle over the full length of the benchmark, and accounting for 22 % of the aggregate power (excluding backlight). All other components showed an RSD of below 3 %.

3.3.4 Phone call

Figure 11 shows the power consumption when making a GSM phone call. The benchmark is trace-based, and includes loading the dialer application, dialing a number, and making a 57-second call. The dialled device was configured to automatically accept the call after 10 sec-

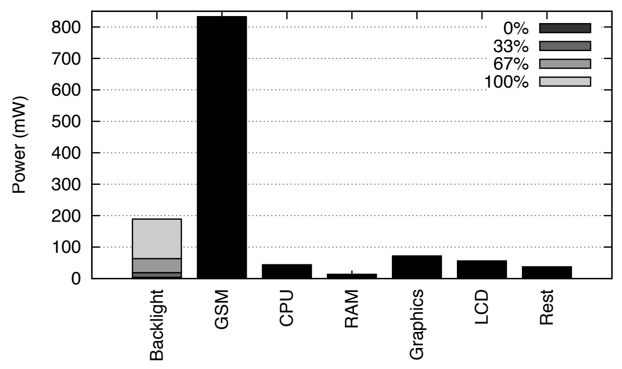


Figure 11: GSM phone call average power. Excluding backlight, the aggregate power is 1054.3 mW.

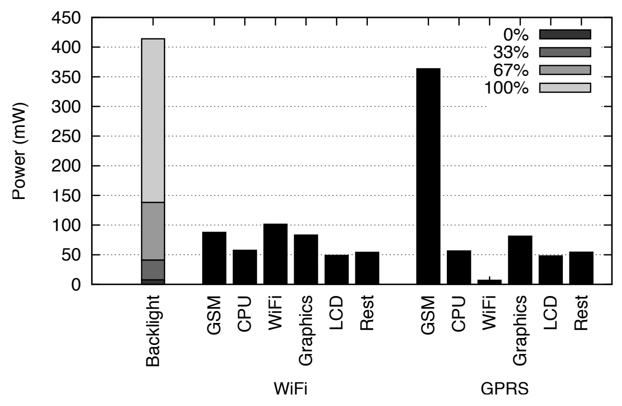


Figure 12: Power consumption for the email macro-benchmark. Aggregate power consumption (excluding backlight) is 610.0 mW over GPRS, and 432.4 mW for WiFi.

onds. Thus, the time spent in the call was approximately 40 seconds, assuming a 7-second connection time. The total benchmark runs for 77 seconds.

GSM power clearly dominates in this benchmark at 832.4 ± 99.0 mW. Backlight is also significant, however note that its average power is lower than in other benchmarks, since Android disables the backlight during the call. The backlight is active for approximately 45 % of the total benchmark.

3.3.5 Emailing

For this benchmark, we used Android's email application to measure the cost of sending and receiving emails. The workload consisted of opening the email application, downloading and reading 5 emails (one of which included a 60 KiB image) and replying to 2 of them. The results of the benchmark are shown in Figure 12, averaged over 10 iterations.

The power breakdown between the GPRS and WiFi

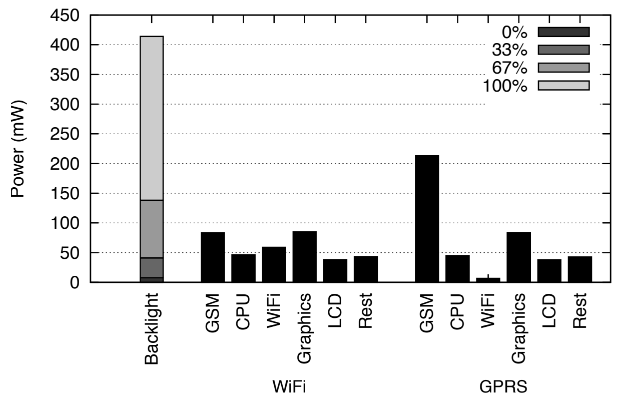


Figure 13: Web browsing average power over WiFi and GPRS. Aggregate power consumption is 352.8 mW for WiFi, and 429.0 mW for GPRS, excluding backlight.

benchmarks is comparable, except for the GSM and WiFi radios. Despite presenting identical workloads to the radios, GSM consumes more than three times the power of WiFi.

3.3.6 Web browsing

Our last benchmark measured the power consumption for a web-browsing workload using both GPRS and WiFi connections. The benchmark was trace-based, ran for a total of 490 seconds, and consisted of loading the browser application, selecting a bookmarked web site and browsing several pages. We used the BBC News website, which we mirrored locally to improve the reliability of the benchmark. After each run, the browser cache was cleared. The results averaged over 10 iterations are shown in Figure 13, including backlight power at 4 brightness levels.

GPRS consumes more power than WiFi by a factor of 2.5. The other components do not display any significant difference between the two benchmarks.

This benchmark, along with the emailing benchmark, are the only two where a more modern phone can be expected to show significantly different results. The much higher bandwidth supported by 3G protocols is likely to result in them being more power-hungry.

4 Validation

In this section, we measure the power consumption of two additional smartphones; the HTC Dream (G1), and the Google Nexus One (N1). Table 6 lists the key features of these devices.

We measure the full-system power of these platforms at the battery; per-component measurements are not possible because the necessary documentation (schematics,

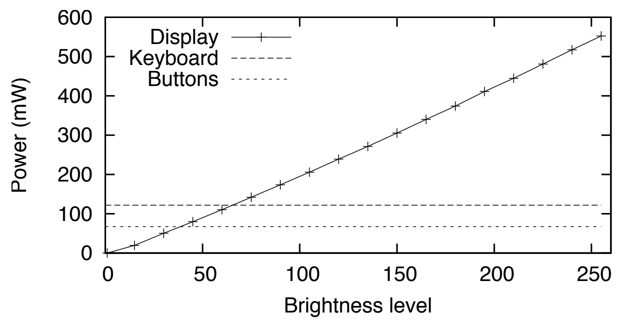


Figure 14: Display, button and keyboard backlight power on the G1.

etc.) are not available to us. Moreover, there is no reason to expect these production devices would be capable of the type of instrumentation we have performed on the Freerunner, since the additional components and PCB area would increase the per-unit cost.

4.1 Display and backlight

Figure 14 plots the power consumption of the various backlights on the G1 as a function of brightness level. In addition to the LCD display backlight, the G1 features a backlit physical keyboard and buttons which are not present on either the Freerunner or the N1. These backlights do not have any brightness control, and contribute 189 mW when both enabled. The content of the LCD display can affect power consumption by up to 17 mW.

The Nexus One features an OLED display, and as such does not require a separate backlight like the Freerunner and G1. Furthermore, the effects of display content and brightness on power consumption are more tightly coupled. For instance, the OLED power consumption for a black screen is fixed, regardless of the brightness setting. For a completely white screen at minimum brightness, an additional 194 mW is consumed, and at maximum brightness, 1313 mW.

4.2 CPU

Figure 15 plots the G1 and N1 total system power under our SPEC CPU2000 workloads at the minimum and maximum frequencies supported by the respective device: 246 MHz and 384 MHz on the G1, and 245 MHz and 998 MHz on the N1. Table 7 shows the percentage slowdown, and reduction in full system power, due to frequency scaling. This benchmark was run with the display system powered down and all radios disabled.

	G1	N1
SoC	Qualcomm MSM7201	Qualcomm QSD 8250
CPU	ARM 11 @ 528 MHz	ARMv7 @ 1 GHz
RAM	192 MiB	512 MiB
Display	3.2” TFT, 320x480	3.7” OLED, 480x800
Radio	UMTS+HSPA	UMTS+HSPA
OS	Android 1.6	Android 2.1
Kernel	Linux 2.6.29	Linux 2.6.29

Table 6: G1 and Nexus One specifications.

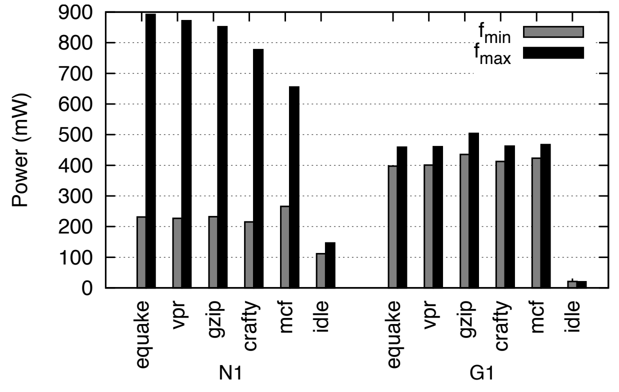


Figure 15: N1 and G1 system power for SPEC CPU2000 benchmarks.

	Performance (%)		Power (%)	
Benchmark	G1	N1	G1	N1
`equake`	67	25	87	26
`vpr`	68	25	87	26
`gzip`	71	25	86	27
`crafty`	76	25	89	28
`mcf`	84	54	91	41

Table 7: SPEC CPU2000 performance and average system power of 246 MHz relative to 384 MHz on the G1, and 245 MHz relative to 998 MHz on the N1.

4.3 Bluetooth

As noted earlier, we were unable to get Bluetooth working reliably on the Freerunner phone. To get an idea of Bluetooth power consumption, we re-ran the audio benchmark on the G1 with the audio output to a Bluetooth stereo headset. The power difference between this and the baseline audio benchmark should yield the consumption of the Bluetooth module, because (as shown in our Freerunner benchmarks) the power consumed by the audio subsystem is almost entirely static.

	Power (mW)	
Benchmark	Total	Bluetooth
Audio baseline	459.7	-
Bluetooth (near)	495.7	36.0
Bluetooth (far)	504.7	44.9

Table 8: G1 Bluetooth power under the audio benchmark.

Table 8 shows the total and estimated Bluetooth power consumption for the audio benchmarks. In the ”near” benchmark, the headset was placed approximately 30 cm from the phone, and about 10 m in the ”far” benchmark.

4.4 Benchmarks

Table 9 shows total system power consumption for the Freerunner, G1, and Nexus One for a selection of our benchmarks. The power consumption of the backlight (OLED for the N1) has been subtracted out, since it is highly dependent on the user’s brightness setting. Table 10 shows the additional power consumption of the OLED display at minimum and maximum brightness levels.

The lower power consumption of the G1 in the idle, web and email benchmarks can be attributed to the excellent low-power state of its SoC and effective use of it by software. This can be seen in the SPEC benchmarks, where the idle system consumes less than 22 mW; the idle CPU power must be lower still.

The power disparity for the phone call benchmark is likely due to power consumed by the non-radio components of the system. The G1 and Nexus One phones enter a suspended state during the call, offloading all functionality to the UMTS module. In contrast, the Freerunner remains in a fully-active state throughout. The power consumption of the GSM subsystem alone (832.4 mW) is comparable to the G1 and N1 system consumption. Due to lack of freely-available documentation, it is not clear whether the Freerunner’s GSM chipset lacks this feature, or if it is not supported in software.

	Average System Power (mW)		
Benchmark	Freerunner	G1	N1
Suspend	103.2	26.6	24.9
Idle	333.7	161.2	333.9
Phone call	1135.4	822.4	746.8
Email (cell)	690.7	599.4	-
Email (WiFi)	505.6	349.2	-
Web (cell)	500.0	430.4	538.0
Web (WiFi)	430.4	270.6	412.2
Network (cell)	929.7	1016.4	825.9
Network (WiFi)	1053.7	1355.8	884.1
Video	558.8	568.3	526.3
Audio	419.0	459.7	322.4

Table 9: Freerunner, G1 and N1 system power (excluding backlight) for a number of micro- and macro-benchmarks.

5 Analysis

5.1 Where does the energy go?

Our results show that the majority of power consumption can be attributed to the GSM module and the display, including the LCD panel and touchscreen, the graphics accelerator/driver, and the backlight.

In all except the GSM-intensive benchmarks, the brightness of the backlight is the most critical factor in determining power consumption. However, this is a relatively simple device from a power-management perspective, and largely depends on the user’s brightness preference. Our results confirm that aggressive backlight dimming can save a great deal of energy, and further motivates the inclusion of ambient light and proximity sensors in mobile devices to assist with selecting an appropriate brightness. Moreover, the N1 OLED results show that merely selecting a light-on-dark colour scheme can significantly reduce energy consumption.

The GSM module consumes a great deal of both static and dynamic power. Merely maintaining a connection with the network consumes a significant fraction of total power. During a phone call, GSM consumes in excess of 800 mW average, which represents the single largest power drain in any of our benchmarks. Unfortunately, a phone-call-heavy workload presents little scope for software-level power management. Dimming the backlight during a call, as Android does, is clearly good policy, saving up to 40 % power even with the large GSM consumption.

Overall, the static contribution to system power consumption is substantial. In all of our usage scenarios, except GSM phone call, static power accounts for at least 50 % of the total. If the backlight is included, this figure rises substantially. This leads us to the conclusion that the most effective power management approach on mobile devices is to shut down unused components and disable their power supplies (where possible).

	OLED Power (mW)	
Benchmark	Min.	Max.
Idle	38.0	257.3
Phone call	16.7	112.9
Web	164.2	1111.7
Video	15.1	102.0

Table 10: Additional power consumed by the N1 OLED display at maximum and minimum brightness.

The RAM, audio and flash subsystems consistently showed the lowest power consumption. While our micro-benchmarks showed that the peak power of the SD card could be substantial (≈ 50 mW), in practice the utilisation is low enough such that on average, negligible power is consumed. Even video playback, one of the more data-intensive uses of mobile devices, showed SD power well under 1 % of total power. RAM has similar characteristics; micro-benchmarks showed that RAM power can exceed CPU power in certain workloads, but in practical situations, CPU power overshadows RAM by a factor of two or more. Audio displayed a largely static power consumption in the range of 28–34 mW. Overall, RAM, audio and SD have little effect on the power consumption of the device, and therefore offer little potential for energy optimisation.

5.2 Dynamic voltage and frequency scaling

Our CPU micro-benchmarks show that dynamic voltage and frequency scaling (DVFS) can significantly reduce the power consumption of the CPU. However, this does not imply reduced energy overall, because the run-time of the workload also increases. Our results show (Table 3) that only highly memory-bound workloads (namely `mcf`) exhibit a net reduction in CPU/RAM energy.

However, such a simplistic analysis assumes that after completing the task, the device consumes zero power. Clearly this is not a realistic model, particularly for a smartphone. To correct for this, we can “pad” each of the measurements with idle power [5] in order to equalise the run times, according to the following equation:

$$E = Pt + P_{\mathrm{idle}}\,(t_{\max} - t)$$

where

E is the equivalent energy consumed for the benchmark;

	% Energy		
Benchmark	Freerunner	G1	N1
`equake`	95.5	126.0	75.6
`vpr`	95.8	124.5	75.9
`gzip`	95.8	120.1	77.7
`crafty`	95.5	115.6	77.3
`mcf`	94.9	105.3	65.9

Table 11: SPEC CPU2000 percentage total system energy consumption of the minimum frequency compared with the maximum frequency, padded with idle power.

P	is the average power over the run-time of the benchmark;
t	is the run-time of the benchmark;
P_{idle}	is the idle power;
$t_{\max}$	is the maximum run-time of the benchmark over all frequencies.

Table 11 shows the energy consumed for each of the SPEC benchmarks at the lowest frequency, compared to the highest frequency, padded with idle power.

The results show that the practical benefits of DVFS depend largely on the CPU hardware (particularly idle power), and to some extent, the workload.

On the G1, which has a good low-power idle mode, reducing frequency always results in increased energy usage. It appears that DVFS on this platform is completely ineffective.

On the Freerunner, DVFS only yields a marginal energy reduction of approximately 5 %—a saving of at most 20 mW. However, the N1 shows considerable advantages to using DVFS, saving up to 35 %, corresponding to an average power reduction of 138 mW. Whether or not to use DVFS on these two platforms is a policy decision, since reducing frequency can affect user experience.

Much of the energy reduction on the Freerunner can be attributed to the high idle power. For a system going into suspend (rather than idle) after completing the workload, DVFS no longer offers an advantage. However, on the N1 this is not the case: DVFS is still effective, even if transitioning into a very-low power state. This is due to the processor's high efficiency at low frequencies, which can be seen in Figure 15.

In the case of an idle system, reducing frequency can result in an energy saving, and at worst has no effect. Our results show that DVFS reduces idle CPU/RAM consumption by about 30 % on the Freerunner. However, in absolute terms, this is less than a 20 mW saving: 6.5 % of an idle system. On the N1, this saving is approximately 36 mW. On the G1, frequency scaling during idle periods is ineffective due to the processor's low-power idle state, which is used aggressively.

5.3 Energy model

We can express the results of Section 3 in a scenario-based energy model of the Freerunner device, which shows the energy for each usage scenario as a function of time:

$$\begin{aligned}
E_{\text{audio}}(t) &= 0.32W \times t \\
E_{\text{video}}(t) &= (0.45W + P_{\text{BL}}) \times t \\
E_{\text{sms}}(t) &= (0.3W + P_{\text{BL}}) \times t \\
E_{\text{call}}(t) &= 1.05W \times t \\
E_{\text{web}}(t) &= (0.43W + P_{\text{BL}}) \times t \\
E_{\text{email}}(t) &= (0.61W + P_{\text{BL}}) \times t
\end{aligned}$$

The equations give the energy consumed in Joules when the time is supplied in seconds. P_{BL} is the backlight power (in watts), scenarios without a P_{BL} term are assumed to run with backlight off.

5.4 Modelling usage patterns

To investigate day-to-day power consumption of the device, we define a number of usage patterns. *Suspend* represents the baseline case of a device which is on standby, without placing or receiving calls or messages. The *casual* pattern represents a user who uses the phone for a small number of voice calls and text messages each day. *Regular* represents a commuter with extended time of listening to music or podcasts, combined with more lengthy or frequent phone calls, messaging and a bit of emailing. The *business* pattern features extended talking and email use together with some web browsing. Finally, the *PMD* (portable media device) case represents extensive media playback. The parameters of these patterns are summarised in Table 12. In each case, GPRS is used for data networking.

The Freerunner uses a battery of 1.2 Ah capacity, which is approximately 16 kJ. Table 13 shows the power use, and resulting battery life corresponding to the above use patterns. We assume that in all cases requiring backlight, illumination level is set at approx 66 %, corresponding to 140 mW. In all other cases, backlight is assumed off.

The table shows that total battery life varies by almost a factor of 2.5 between use cases. It shows that GSM is the dominating energy drain, followed by CPU and graphics.

Workload	SMS	Video	Audio	Phone call	Web browsing	Email
Suspend	-	-	-	-	-	-
Casual	15	-	-	15	-	-
Regular	30	-	60	30	15	15
Business	30	-	-	60	30	60
PMD	-	60	180	-	-	-

Table 12: Usage patterns, showing total time for each activity in minutes.

	Power (% of total)							Battery life
Workload	GSM	CPU	RAM	Graphics	LCD	Backlight	Rest	[hours]
Suspend	45	19	4	13	1	0	19	49
Casual	47	16	4	12	2	3	16	40
Regular	44	14	4	14	4	7	13	27
Business	51	11	3	11	4	11	10	21
PMD	31	19	5	17	6	6	14	29

Table 13: Daily energy use and battery life under a number of usage patterns.

5.5 Limitations

Our work has a number of limitations which need to be kept in mind when using our results.

The biggest one is that the Freerunner is not a latest-generation mobile phone, but is a few years old. The main feature it is lacking is a 3G cellular interface, which supports much higher data rates than the 2.5G GPRS interface. Our validation results show that this higher data rate does not appreciably affect power consumption in practical situations.

Further, the application processor is based on a relatively dated ARMv4 architecture, however it is clocked at a rate consistent with 2009-vintage smartphones. The difference in power consumption compared with more modern processors can traced largely to idle power; in other respects, the age of the CPU is not a substantial limitation.

6 Related Work

Mahesri and Vardhan [4] perform an analysis of power consumption on a laptop system. Their approach to component power measurement is driven partially by direct power measurement, but largely by deduction using modelling and off-line piece-wise analysis. They show that the CPU and display are the main consumers of energy for their class of system, and that other components contribute substantially only when they are used intensively. Their results mirror our observations that RAM power is insignificant in real workloads.

Bircher and John [2] look at component power estimation using modelling techniques. They demonstrate an error of less than 9 % on average across all tested subsystems, including memory, chipset, disk, CPU, and I/O.

In a later work, Bircher and John [3] measure the power consumption of the CPU, memory controller, RAM, I/O, video and disk subsystems under a number of workloads. Their results show that CPU and disk consume the majority of the power, with the RAM and video systems consuming very little. However, under the SPEC CPU suites, they show that RAM power can indeed exceed CPU power for highly memory-bound workloads.

Sagahyroon [8] perform an analysis similar to ours on a handheld PC. They show significant consumption in the display subsystems, particularly in backlight brightness. Unlike our results, theirs suggest that the CPU, and its operating frequency, is important to overall power consumption. They also show significant dynamic power consumption in the graphics subsystems.

7 Conclusions and Future Work

We performed a detailed analysis of energy consumption of a smartphone, based on measurements of a physical device. We showed how the different components of the device contribute to overall power consumption. We developed a model of the energy consumption for different usage scenarios, and showed how these translate into overall energy consumption and battery life under a number of usage patterns.

The open nature of the Openmoko Neo Freerunner smartphone is what allowed us to perform such a detailed analysis and breakdown of its power consumption. This is not possible to the same degree on a typical commercial device.

We have compared the detailed measurements with a coarse-grained analysis of more modern phones, and shown the results to be comparable.

The ultimate aim of this work is to enable a systematic approach to improving power management of mobile devices. We hope that by presenting this data, we will enable such future research, both in our lab as well as by others.

Acknowledgments

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.

Thanks to Nicholas FitzRoy-Dale, who provided us with the input event capture and replay tools, and Yanjin Zhu, who allowed us to run measurements on her Nexus One. We would also like to thank Bernard Blackham, Etienne Le Sueur, Leonid Ryzhyk and our anonymous reviewers for their feedback on earlier versions of the paper.

Availability

Relevant software and data is available at `http://ertos.nicta.com.au/software/`.

References

[1] ANDRIOD ON FREERUNNER COMMUNITY. 2009. `http://code.google.com/p/android-on-freerunner/`.

[2] BIRCHER, W. L., AND JOHN, L. K. Complete system power estimation: A trickle-down approach based on performance events. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (San Jose, CA, USA, Apr. 25–27 2007), IEEE Computer Society, pp. 158–168.

[3] BIRCHER, W. L., AND JOHN, L. K. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd International Conference on Supercomputing* (Island of Kos, Greece, June 2008), pp. 327–338.

[4] MAHESRI, A., AND VARDHAN, V. Power consumption breakdown on a modern laptop. In *Proceedings of the 2004 Workshop on Power-Aware Computer Systems* (Portland, OR, USA, Dec. 2004), B. Falsafi and T. N. Vijaykumar, Eds., vol. 3471 of *Lecture Notes in Computer Science*, Springer, pp. 165–180.

[5] MIYOSHI, A., LEFURGY, C., HENSBERGEN, E. V., RAJAMONY, R., AND RAJKUMAR, R. Critical power slope: understanding the runtime effects of frequency scaling. In *Proceedings of the 16th International Conference on Supercomputing* (New York, NY, USA, June 2002), ACM Press, pp. 35–44.

[6] NATIONAL INSTRUMENTS CORPORATION. *NI 622x Specifications*, June 2007. 371290G-01.

[7] OPENMOKO, INC. GTA02 — Neo Freerunner schematics. `http://downloads.openmoko.org/developer/schematics/GTA02`, Aug. 2008. A7RC0.

[8] SAGAHYROON, A. Power consumption in handheld computers. In *Proceedings of the International Symposium on Circuits and Systems* (Dec. 2006), pp. 1721–1724.

[9] SNOWDON, D. C., LE SUEUR, E., PETTERS, S. M., AND HEISER, G. Koala: A platform for OS-level power management. In *Proceedings of the 4th EuroSys Conference* (Nuremberg, Germany, Apr. 2009).

[10] U-BLOX AG. *ATR0630 Data Sheet*, July 2006. GPS.G4-X-06009-P2.

[11] WIKIPEDIA. Smartphone. `http://en.wikipedia.org/wiki/Smartphone`. Last visited January 2010.

SleepServer: A Software-Only Approach for Reducing the Energy Consumption of PCs within Enterprise Environments

Yuvraj Agarwal *Stefan Savage* *Rajesh Gupta*

Computer Science and Engineering
University of California, San Diego
{*yuvraj,savage,gupta*}*@cs.ucsd.edu*

Abstract

Desktop computers are an attractive focus for energy savings as they are both a substantial component of enterprise energy consumption and are frequently unused or otherwise idle. Indeed, past studies have shown large power savings if such machines could simply be powered down when not in use. Unfortunately, while contemporary hardware supports low power "sleep" modes of operation, their use in desktop PCs has been curtailed by application expectations of "always on" network connectivity. In this paper, we describe the architecture and implementation of SleepServer, a system that enables hosts to transition to such low-power sleep states while still maintaining their application's expected network presence using an on-demand proxy server. Our approach is particularly informed by our focus on practical deployment and thus SleepServer is designed to be compatible with existing networking infrastructure, host hardware and operating systems. Using SleepServer does not require any hardware additions to the end hosts themselves, and can be supported purely by additional software running on the systems under management. We detail results from our experience in deploying SleepServer in a medium scale enterprise with a sample set of thirty machines instrumented to provide accurate real-time measurements of energy consumption. Our measurements show significant energy savings for PCs ranging from 60%-80%, depending on their use model.

1 Introduction

"Turn off lights and equipment when they are not in use." This simple exhortation heads the list of the Environmental Protection Agency's "tips" for making businesses energy efficient. The reasons are straightforward. In the U.S., commercial buildings consume over one third of all electrical power [9] and, of these, lighting and IT equipment are the largest contributors (roughly 25% and 20% respectively in office buildings according to one 2005 study [10]). However, while it has been relatively straightforward to address lighting use (either through education or occupancy sensors), IT equipment use has been far more resistant to change. Indeed, in a recent empirical study across a number of buildings on our campus, we measured that between 50% and 80% of all electrical power consumption in a modern building is attributable to IT equipment (primarily desktops) [4].

This finding can be unintuitive. First, the computer equipment industry is working hard to reduce power consumption at all levels. Thus, we expect desktop power consumption to be decreasing, not increasing. Second, modern hardware and operating systems possess mechanisms for entering low-power modes when not in use. However, the overall impact of both has been limited in practice. For example, while individual components are indeed much more energy efficient, the capability per desktop has also increased. Thus, while a typical desktop system from 2002 might consume roughly 60-75 watts when idle, the same is also true for today's desktops. Even machines designed and marketed as "low-power" desktops, such as Dell's Optiplex 960 SFF, routinely consume 45 Watts when they are unused.

Compounding this issue is the fact that while today's machines *can* enter a low-power sleep state, it is common that they do not – even when idle. Here the problem is more subtle. Today's low power mechanisms assume that – like lighting – the absence of a user is a sufficient condition for curtailing operation. While this is largely true for disconnected laptop computers (indeed, low-power suspend states are more frequently used for such computers), it is not compatible with how users and programs expect their connected desktops to function. The success of the Internet in providing global connectivity and hosting a broad array of services has implicitly engendered an "always on" mode of computation. Applications expect to be able to poll Internet services and download in the background, users expect stateful applications to act on their behalf in their absence, system administrators expect to be able to reach desktops remotely for maintenance and so on. Thus, there is implicitly a high "opportunity cost" associated with not being able to access and use computers on demand.

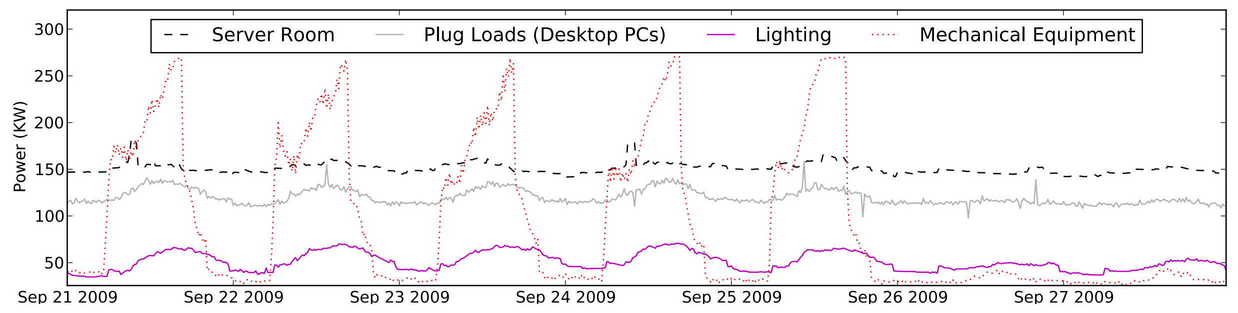


Figure 1: *Detailed breakdown of the various power consumers inside the CSE building at UC San Diego [4], for a week in September 2009. Desktop computing equipment, which make up majority of the plug loads, and the IT equipment in the server rooms account for almost 50% to 80% of the base load of this building.*

However, we, as well as others, have observed that this demand for "always on" *behavior* is distinct from truly requiring an "always on" system. Indeed, it can be sufficient to present the *illusion* that a desktop is always on — suspending it when idle, proxying minor requests on its behalf and dynamically waking it up if its power and state are truly needed [3, 5, 16, 20]. Unfortunately, all of these systems have imposed significant barriers to deployment in implementing this illusion — either requiring significant modifications to network interface hardware and in some cases the host OS software. Such requirements not only represent new expenses, but also require the active participation of third parties (especially network silicon merchants) over the full range of systems in broad use. Thus, in spite of the significant potential for energy savings, we are unaware of any such systems that have been fielded in practice or had practical impact on power consumption in the enterprise sector.

In this paper, we focus on this deployment challenge. Our goal is to provide the same power savings of prior research prototypes, such as our own Somniloquy system [3], yet do so within the confined rubric of existing commodity network equipment, host hardware, operating systems and applications. Indeed, at the *idea level* SleepServer is similar to Somniloquy, but from a practical standpoint SleepServer addresses a different set of challenges that arise directly from our experience in deploying it in our department. The remainder of this paper describes our two contributions: First, we motivate and explain the architecture and implementation of the SleepServer system, which transitions machines to a sleep state when idle, while transparently maintaining the "always-on" abstraction should their services be needed, using a combination of virtual machine proxies and VLANs. Second, we present the results of our pilot SleepServer deployment across a heterogeneous sample of thirty desktops in active use and monitored in real-time by dedicated power meters, including an empirical quantification of the (significant) power savings, an analysis of our system's scalability and cost, and a description of our qualitative experience concerning user feedback and behavior modification.

2 Background

Over the last several decades, the pervasive adoption of information technology has created a new demand for electrical power. Partly due to this reason, the share of U.S. electrical power consumed by commercial buildings has grown 75% since 1980 (it is now over a third of all electrical power consumed) [9]. In a modern office building, this impact can be particularly striking.

For example, Figure 1 shows the power consumption of the CSE building at UC San Diego, broken down by various functions: lighting, server computing, plug loads and HVAC. While the overall electrical usage varies from 320KW to 580KW over the course of a year [4] — generally due to increases in air-handling and cooling — the baseline load is highly stable. Indeed, computer servers and desktop machines connected as plug loads account for 50% (during peak hours on weekdays) to 80% (during nights and weekends) of the baseline load and vary by no more than 20KW over the course of the year.

Given their large consumption footprint, it is not surprising that increasing the energy efficiency of IT equipment has long been an active area of research. However, most of these efforts fall into three distinct categories. One approach focuses on reducing the active power consumption of individual computing devices by utilizing lower power components [11] or using them more efficiently [12, 21]. The second class of energy saving techniques, especially popular in data centers, look at migrating work between machines — either to consolidate onto a smaller number of servers [8] (e.g., using virtual machines [19]) or to arbitrage advantageous energy prices in different geographic zones [23]. Finally, the third class of energy management techniques consider opportunistically duty-cycling subsystems, such as wireless radios

[2, 22, 24], networking infrastructure [14, 21] or even entire platforms[3, 18, 25], during periods of idleness or low use. SleepServer falls into this third category of energy management approaches.

The duty-cycling technique exploits the capability of modern hardware to enter low-power states while maintaining transient state. For example, modern desktops support the ACPI S3 (Sleep/Standby) state, which can reduce power consumption by 95% or more [1]. One approach to using this capability, embodied in modern versions of most operating systems, is to simply place the system in a low-power state after it has been idle for some period of time. Unfortunately, as mentioned earlier, this conflicts with the behavior of users and software that implicitly assume an "always on" abstraction.

To manage this problem, today's network interfaces (NIC) implement features, such as "Wake-on-Lan" [17], that allow sleeping systems to be awakened upon receiving network traffic (frequently a special packet). While this mechanism is quite important, it does not address the key question of *when* a machine should be woken. If this mechanism is activated for every packet then energy savings quickly disappear. Conversely, if its use is too restricted then the "always on" abstraction is lost and users lose the ability to freely access their machines. Consequently, a gap exists between the abstraction levels over which WoL works and the level at which its operation is useful in real-life systems.

Some recent variants have attempted to address these concerns through proprietary hardware and software support. For example, Intel Remote Wake allows the "wake up" capability to be integrated into server software so, for example, a VoIP server could be enabled to wake one of its client machines [15]. Apple's Wake-on-Demand takes a similar approach, allowing client machines using Bonjour advertised services to be "woken" when accessed via Apple networking hardware (WiFi APs) [6]. While neither approach is general, they reflect precisely the need to encode some dynamic triggering policy to preserve application and user transparency.

To generalize this policy, several systems incorporate additional low-power processors into the network interface itself[3, 25]. Using this approach, requests from the network can be parsed and evaluated even when the rest of the system itself is in a low-power sleep state. Moreover, due to their generality these low-power processors can even process requests on behalf of the sleeping system instead of waking it, thus maximizing the amount of power saved. Unfortunately, such approaches face a significant deployment barrier as they require non-trivial changes to network interface hardware.

Finally, a set of projects [5, 16, 20] have explored the notion of implementing this "always on" functionality via network *proxies* that maintain a limited network presence on behalf of sleeping PCs. Nedevschi et al.[20] provide an in-depth look at network traffic to evaluate the design space of a network proxy, while the Network Connection Proxy (NCP) [16] proposes modifications to the socket layer for keeping TCP connection alive through sleep and resume transitions. SleepServer is most similar in spirit to these efforts, but is distinguished from prior work both in offering an actual implementation and not requiring changes to existing hardware, software or networking infrastructure. We argue that these are necessary requirements for any system to see practical use in the enterprise setting.

3 SleepServer: Architecture

We had several goals in mind when we started to design a network-proxy, especially for an enterprise setting. First, the proxy must be able to maintain the network presence of any host on the local network while maintaining complete transparency to other end hosts in the network and to network infrastructure elements such as switches and routers. Second, since the proxies themselves add to the total power consumption, they must be highly scalable and therefore be able to service hundreds of hosts at any given time for maximum energy savings. Third, the proxy should be able to provide isolation when it is servicing individual hosts while providing mechanisms to scale resource allocation based on the proxying demands of individual hosts. Fourth, the proxy must address management aspects, such as providing mechanisms to enable and disable the proxying functionality for hosts dynamically, viewing the status of supported hosts in the system, and maintaining security. Fifth, the proxy should be able to support a heterogeneous environment with different classes of machines running different operating systems. Lastly, we wanted to achieve all of the above goals purely in software without requiring any additional hardware to the individual end hosts or any changes to the networking infrastructure.

Based on these design goals, our SleepServer—network-proxy architecture is illustrated in Figure 2. In an enterprise LAN environment, one or more SleepServers (SSR) can be added in addition to the host computers (H) proxied by the SleepServer. These SleepServer machines have a presence on the same network segments or subnets as the proxied hosts, i.e. they are on the same Layer-2 broadcast domain. A SleepServer can proxy for machines on different subnets using existing Virtual LAN (VLAN) support that is common to commodity routers and switches. Of course, there can be multiple SleepServers, each servicing only a particular VLAN(s) for security isolation if required by enterprise policy.

The various components of a SleepServer are shown

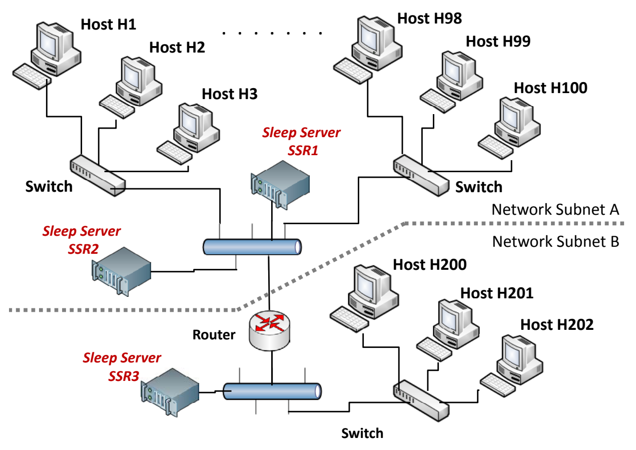


Figure 2: *An example deployment of Sleep Servers in an enterprise setting. Since there are many more hosts in Subnet A there may be more than one SleepServer (SS1 and SS2) to handle the load while there is only one SleepServer (SS3) needed in Subnet B with fewer hosts.*

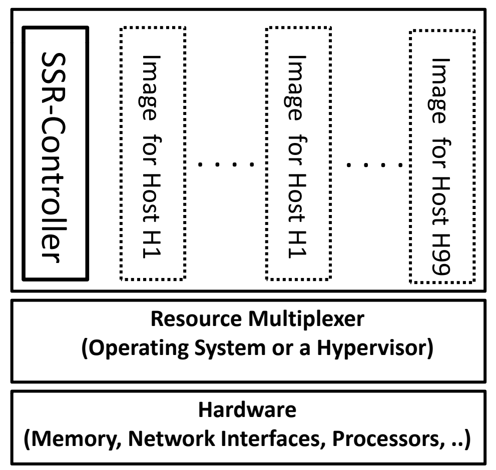


Figure 3: *A example SleepServer serving a collection of host PCs (H1, ...H99). All resource sharing and access to the hardware is mediated by the SleepServer controller software module running on the SleepServer.*

in Figure 3. As shown in the figure, access to the underlying hardware is by a resource multiplexer, which can be either an operating system or a hypervisor/Virtual Machine Monitor (VMM) such a XEN [7]. For each host H that the SleepServer is proxying for, there is a corresponding *Image* I that is instantiated. This image is responsible for maintaining the network presence of the host when it is in a sleep state. Although it is possible to build a host image as a stand alone process that can respond to the various network protocols, we chose a VMM-based architecture for simplicity and expediency. Since VMs are typically based on existing operating systems, all the standard protocols (e.g. ARP, ICMP) are already supported while support for others can be easily added. In contrast a process based approach would require adding support for the myriad of standard protocols. Furthermore, it is unclear how a process oriented approach would handle stateful applications that need application specific code (described in Section 3.2). VMMs also already have existing support for isolation between host images, for resource allocation and sharing, and for managing security and networking between images. While VMs may use more resources than a standalone process, our results show that the VM solution offers sufficient scalability for our purposes without requiring significant additional engineering. In addition to the host images, the SleepServer supports a privileged *controller* domain that is responsible for various SleepServer functions. This SSR-controller manages the creation and configuration of individual host images, communication between the SSR-Client software and the host images, and resource allocation and sharing among the host images.

Each host PC using SleepServer has a software component installed (SSR-Client) that communicates with the SSR-Controller. When a particular host is enabled for use with a SleepServer, the SSR-Client first connects to the SleepServer machine in its network subnet, and specifies its network parameters such as its MAC and IP address and its firewall configurations. In addition, the SSR-Client sends the state of the running applications on the host, and any open TCP or UDP ports to the SSR-Controller. The information sent by the host is received by the SSR-Controller which then creates an 'image' of that particular host using the specified network parameters. The network parameters of this image are configured to mimic those of the particular host. The base firewall configuration of this image can be identical to the one on the host, or can be made more restrictive. When the host is asleep, its image can respond to incoming packets on its behalf. In case an application request is received that requires the host itself to respond the SSR-Controller wakes up the host and disables its image on the SleepServer.

3.1 Handling State Transitions

The basic operation for a SleepServer is as follows. Before a host PC transitions to a low power mode such as sleep, the SSR-Client software running on it sends a message to the SSR-controller with the state transition information. The controller then enables the corresponding Image for that host. Additionally, in order to have packets re-routed to the SleepServer and the Image of the host, the controller needs to reconfigure the Layer-2 switches to re-learn the network topology. To do this in a seamless way, without requiring any special functionality provided by only high-end switches, the SleepServer uses a combination of gratuitous ARPs and packets sent to the gateway in the subnet. Since these packets are sent by the Images of the host on the SleepServer, the Layer-2 switches learn the MAC address of the Image and subsequent packets for that host are sent to the switch ports that the SleepServer is connected to.

Similarly, when the host transitions out of a low power mode, the SSR-Client traps this event and sends a message to the SSR-Controller notifying it of the transition. When the SSR-Controller gets this message it disables the host image, thus stopping if from responding on the behalf of the host. The SSR-Client on the host also sends gratuitous ARP messages and packets to the subnet gateway which cause switches in the network infrastructure to learn the MAC address and forward any subsequent packets meant for the the host to the switch port that the host is connected to.

3.2 Host Images on the SleepServer

The *host-images* on the SleepServer are responsible for maintaining full network presence on behalf of the host when they are asleep. In principle, these host-images have their own TCP/IP stack, memory, processor resources and persistent storage. The host images do not need to run the same OS as the host computer. The image of a particular host is configured with the identical network configuration as the host itself (IP, MAC address) and it can essentially masquerade as the host and respond to network events when the host is asleep. However, processor and memory resources allocated to an image are generally much less than those available on the host machine itself, as these host images are configured to only maintain network presence and any application stubs that may be necessary to run (described later in this section). For example, a host image on the SleepServer may only have 64MB of memory allocated, while the actual host may have several Gigabytes of memory. Furthermore, shared resources such as the processor and network bandwidth allocation of the images are multiplexed between several other images on the same SleepServer, providing scalability and the ability to host hundreds of images on the same SleepServer.

Supporting Stateless Applications: Stateless applications do not maintain long running sessions or have a persistent connection open. To support these applications, the image responds appropriately to connection requests by doing one of two actions. First, the image can respond on behalf of the host for certain requests by sending an appropriate response, such as replying to ICMP requests or responding to ARP queries. Recall that since the network parameters of the image are the same as the host, they appear identical to the other hosts on the network. Second, for incoming requests that require the resources of the host itself, for example an incoming SSH connection to the host or an SMB request for data stored on the host computer, the image is disabled and the controller is notified. To ensure that the original connection request is handled appropriately, it is essential that the image of the host does not respond to it. Instead we rely on the fact that most applications are based on protocols, such as TCP, that normally retry connection requests in case of packet loss. Applications based on unreliable delivery protocols such as UDP usually handle packet loss at the application layer by retransmitting requests. Applications that are essentially stateless and connect to well defined ports, such as remote access requests using RDP (TCP Port 3389) or SSH (TCP Port 22), incoming SMB file sharing requests(TCP port 445), and requests to a web server (TCP Port 80), can be supported using this mechanism.

On receipt of the notification from a host image, the controller automatically generates a wakeup packet to wake up the host. This can be done using either Wake-on-LAN (WoL)[17], which can be found on most PCs, or by utilizing newer technologies like Intel AMT. WoL allows PCs to be woken up on receipt of several different kinds of packets. 'Wake on Directed Packets' and 'Wake-on-Any Packet' unfortunately cause too many wake ups since even broadcast traffic causes the PC to wake up. Instead, we use the "magic-packet" variant of Wake-on-LAN which can be sent by a SleepServer in the subnet to wake up the host from a sleep state.

Supporting Stateful Applications: Stateful applications maintain continuous state and send periodic keep alives or keep connections open. For these applications and protocols, capturing application semantics is essential in order to proxy for them by the image of the host on the SleepServer. To support these stateful application we require application specific code to be running on the images on the SleepServer. This is in contrast to the stateless applications mentioned in the previous section that do not require any application specific code on the images. A majority of these stateful applications run in the background, and can be active even when the user is not present in front of the system. Examples include maintaining presence on IM networks, long running and unattended web downloads, participating on P2P networks such as BitTorrent, and advertising available services and content using protocols such as Bonjour and uPNP.

To support these applications we take an approach similar to Somniloquy[3] where we run reduced functionality variants of the main applications, called 'application-stubs'. These stubs have significantly reduced processor and memory requirements as compared to the original applications running on the hosts. The key idea in developing a stub is to remove all the code components of the application that are not needed on the host image, such as the user interface. Similar to Somniloquy, these stubs can be created by either writing them from scratch or by removing components of an existing application. In some cases console versions of the same applications are already available, such as the `pidgin` IM client and its console version called `finch`, which can be used as a starting point.

Although the process to build stubs is similar to that used in Somniloquy, there are several key differences that make it significantly simpler in the SleepServer architecture. First, because SleepServer can be run on any x86 based server computer, the host images themselves can also run on this industry standard architecture. Therefore, porting applications and building stubs for the SleepServer images is as simple as building an application for a regular computer with all of the standard libraries and packages available. In contrast, Somniloquy used an additional piece of hardware, with a different processor architecture, and required cross compiling applications. Second, the host images running on a SleepServer are based on software VMs, and as a result the amount of resources allocated to each host image can be dynamically changed. For example, the image of a particular host performing heavy downloads would have more memory and processor resources allocated to it, while the image of another host that is just replying to ICMP echo requests would have less. This is not possible with Somniloquy as each host PC has a dedicated piece of Somniloquy hardware physically attached to it, each with a fixed amount of resources.

In some cases it becomes necessary to transfer data between the host and its image running on the SleepServer. For example, consider a download stub that continues a long running download on behalf of the host when it is asleep. Once the host wakes up, the downloaded data needs to be transferred to the host. In the SleepServer architecture this state and data transfer can be handled by storing the data locally in the persistent storage provided to each image and then later sending it to the host over the network when the host is awake. Another option is to set up a network storage for each host, which can even be hosted by the SleepServer itself. The host and its image can then access the same unified storage to store data that is needed for SleepServer operation.

3.3 Scalability and Resource Sharing

Scalability, in terms of the number of hosts supported simultaneously on a single SleepServer, is an important design goal for both cost and energy savings. To keep the cost of the SleepServer low, we want to base the SleepServer on commodity components and have it support a large number of hosts. Therefore, we ensure that the individual images start with the smallest possible footprint, both in terms of disk space used and the number of processes they create, in order to minimize processor and memory usage. Furthermore, each image is further customizable such that only the application stubs or software modules that are needed by each host are loaded onto their respective images.

Beyond CPU usage, the potential scalability bottlenecks lie in the memory usage and network bandwidth requirements. Currently we allocate memory statically to the host images and only have as many images concurrently running as can fit in the main memory of the SleepServer. Given that our images start out with very modest memory allocations (64MB or less), a SleepServer with 32GB of memory can support over 500 simultaneously executing host images. Furthermore, since the host images are based on Virtual Machines (VM), we can employ techniques such as Difference Engine [13] which exploits memory compression techniques to significantly reduce the memory use of VMs. For multiplexing access to the processor and the network interfaces, we rely on the resource sharing provided by the underlying VMM.

3.4 Management in Enterprises

Security and manageability are important considerations in enterprises. System administrators are reluctant to add and support technology solutions that add administrative costs. We have implemented management modules that allow administrators to view in real time a ‘heart-beat’ of the systems supported on SleepServer. Since all state transitions such as hosts going to sleep and resuming are logged by the SleepServer controller, it can also provide users of those particular PCs feedback on their energy usage in real time and their estimated energy savings. SleepServer administrators can check the health of the host machines and see if they are transitioning in and out of sleep modes successfully. SleepServer also adds to the observability of the state of machines. For example, it is possible to tell the difference between a computer in a sleep state against one that has crashed. Through the centralized management interface, administrators can also set up host specific policies, such as waking up some hosts at designated times, and perhaps even staggering wake ups to minimize spikes in energy usage.

Similarly, failure detection and recovery are important, such as handling the case when a SleepServer goes down. Note that under all circumstances the hosts that are sleeping can still be woken up normally by a user action such as a key press on the keyboard. Hosts that are awake and were not being serviced by the SleepServer are not affected by a SleepServer failure, while hosts that were asleep will lose network connectivity. Furthermore, if the SleepServer is unavailable any hosts that want to transition to sleep and maintain their network presence and availability can no longer do so. The SleepServer architecture handles these failure cases using several mechanisms. First, for a temporary failure or an intentional reboot after updates the SSR-controller re-creates the state of the various hosts from its logs and restarts all the host images to their original conditions. Second, multiple SleepServers can exist and proxy for a particular host. The different SSR-Controllers in this case communicate with each other to provide redundancy and load balancing. Finally, hosts can discover and check for the availability of their SleepServer and in case the SleepServer is not responding, they can look for alternatives. If no SleepServer is available the SSR-Client running on the host alerts the user about the lack of an appropriate SleepServer in the network and can let the user decide if they still want to transition to sleep.

Security and Isolation of the Host images: Addressing the security implications of SleepServer is important since multiple host images are hosted on the same SleepServer. We need to ensure that the host images do not increase the attack surface of the hosts within an enterprise while keeping them safe from outside attackers. Furthermore, the individual images should not be able to receive and intercept each others network traffic.

While we do not currently have a comprehensive security evaluation, there are several features and safeguards in our SleepServer architecture that address security. First, since the SleepServer is based on a VMM architecture the SSR-Controller domain runs at a higher privilege level than the individual images. The SSR-Controller therefore has the responsibility to add rules to route traffic to the appropriate image. As such, only the packets that are meant for a particular host image are send to it, in addition to broadcast and multicast traffic. Second, the host images are not accessible by users of the host PCs directly. Instead, all communication between the SSR-Client software and the host image goes through the SSR-Controller. Third, the firewalls on the host images is configured to be very restrictive and opened only to the ports for which the host and its application stubs require. Note that the firewall on the host images can be configured to be even more restrictive than that of the actual host. Fourth, the host images only communicate with the SleepServer controller directly to get configuration changes and can be patched by the controller. Furthermore, we enable only the essential services and programs in the host images, and as such the attack surface is relatively narrow. Finally, recall that on a valid incoming connection request the particular host is immediately woken up from sleep by the controller and its image stops responding. In this case, the security implications are identical to the case where the host remained awake.

4 Implementation

We have implemented SleepServer on a commodity server computer and are currently serving over thirty desktop users on it. In this section we outline our implementation of SleepServer, specifically highlighting how we support the design goals mentioned earlier in Section 3. There are three primary software components that are required. The first is the SSR-Client software that runs on the host computer. The second component is the SSR-Controller which runs on a SleepServer computer. The third component are the host images themselves, each supporting a host PC using SleepServer.

4.1 SS-Client Software for Hosts

SleepServer currently supports several common operating systems, such as Microsoft Windows (XP, Vista and 7) and Linux (tested on Ubuntu). Windows based operating systems have standardized power management interfaces but different distributions of Linux can have different interfaces to handle power management and therefore require different client software.

While SleepServer can support multiple low-power states, in our evaluation we use only the standard ‘sleep’ state or suspend-to-RAM (ACPI State S3) across all machines [1]. In some cases this requires changing the BIOS settings of the host to enable the S3 state. Since we are leveraging Wake-on-LAN functionality, and specifically ‘magic packets’, we require the appropriate options to be enabled in the BIOS, the device drivers and the operating system. Most PCs manufactured in the last decade support S3 and Wake-on-LAN, although these modes often need to be enabled explicitly.

The SSR-Client software on the host computers is responsible for providing mechanisms to detect power management events, such as transitions in and out of sleep modes, and transfer state information to the SSR-Controller such as firewall configuration, network information (IP, MAC addresses), applications and events that the host wants to be notified for.

Note that modern operating systems already have user-configurable power management idle timers that use events, such as keyboard, mouse, and CPU activity to determine when the host is inactive and able to sleep. SleepServer users can use the same interface to configure their idle preferences. It is important to note that almost all of the users in our deployment had these power management timers disabled before using SleepServer since they wanted to be able to access their PCs at all times. In our evaluation we compare the additional energy savings gained by using these automatic idle timeouts, as compared to having no automatic idle timeouts and instead asking users to manually put their machines to sleep.

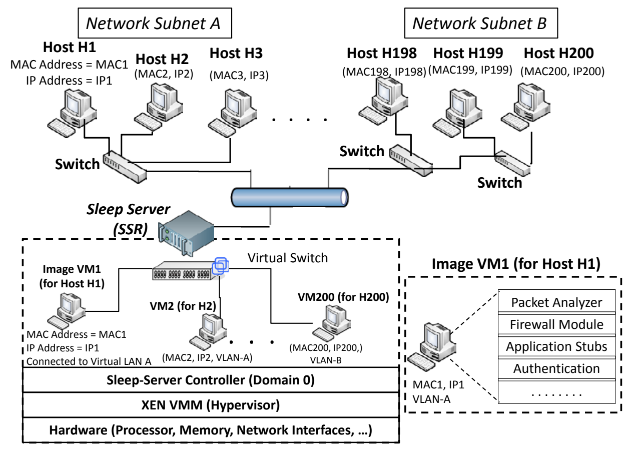


Figure 4: *SleepServer implementation based on XEN.*

Windows Platforms: The SSR-Client for Microsoft Windows is comprised of several programs and services. The first component is an initial setup program that is used to read the firewall configuration of the host PC as well as its network configurations (IP address, Host name, MAC address) and send this information to the SSR-Controller. Anytime these parameters change, this program sends an update to the SSR-Controller. The second component is a 'PowerNotifier' that is responsible for updating the SSR-Controller of any change in the power state of the host. Since there are various ways a user in Windows can transition to sleep modes, the PowerNotifier service installs hooks directly into the Windows Power Management Interface (WMI) so that it is notified on any power state changes. When a suspend or resume from sleep event occurs the PowerNotifier component sends a message to the SSR-Controller.

Linux Platforms: Similar to our implementation for Windows platforms, we have several components that run on the Linux host. The Ubuntu distribution allows access to the power management events through the ACPI subsystem and our PowerNotifier service for Linux is installed by placing appropriate hooks into this ACPI framework. PowerNotifier is called on both sleep and resume from sleep events and is responsible for communicating with the SSR-Controller. Additionally, we use standard Linux tools such as `iptables` and `ifconfig` to get network and firewall configurations.

4.2 SleepServer

We have implemented the SleepServer on a commodity Dell PowerEdge PE2950 server, which is configured with two quad-core Intel XEON 5550 processors, 32GB of RAM, a 1TB SATA disk drive and dual gigabit interfaces. Figure 4 illustrates the logical organization of our SleepServer prototype, and the host images running on it. The SleepServer runs a XEN 4.0 [7] hypervisor/VMM (using the 2.6.32 pvops kernel) and the SSR-Controller (domain0 in XEN) is based on Ubuntu 9.10. We have modified the XEN utilities to allow creation of customized SleepServer Virtual Machines (VM) (domU's in XEN) representing host images based on supplied network parameters such as Host name, IP address, MAC address, etc. We configured the SSR-Controller to have several virtual interfaces that allow it to be placed on all of the VLANs (eight different subnets) in the CSE department network at UCSD. We also configured the department managed switches so that traffic on all these VLANs is forwarded to the switch port that the SleepServer is connected to. The SSR-Controller then sets up software network bridges automatically for each configured VLAN. We initialize the VMs to only have access to those VLANs that the host they represent are originally on, with identical network parameters (IP, MAC address etc) as the hosts. For example, image VM1 for Host H1 can only access VLAN A and will have the same IP and MAC addresses as the host H1 (Figure 4). Additionally, the SSR-Controller and the host VMs communicate over a separate private network.

The SSR-Controller listens for messages from the hosts on several well defined UDP and TCP ports. Recall that our SSR-Client software running on the hosts automatically sends state transition messages to the SSR-Controller. On receipt of these messages the SSR-Controller enables (if the host is going to sleep) or disables (if the host is resuming from sleep) the VM for the appropriate host. The SSR-Controller is also responsible for reconfiguring the VM for a particular host when the SSR-Client sends an update, such as adding or deleting new applications. Additionally, the SSR-Controller maintains a log of all sleep/resume events received. These recorded events are used by a separate status module which allows users to view the status of their PCs. This log is also key in calculating the duty-cycle of all the hosts served on the SleepServer, and can provide estimates of energy savings given the average power draw of the machine in sleep and active modes. The SSR-Controller has a wake-up module that is used to generate wakeup packets using Wake-on-LAN to resume a sleeping host when needed. Note that since both the SSR-Controller and the wakeup-module have a presence on all the VLAN's, they can send Wake-on-LAN magic packets on any department subnet (same layer-2 domain) without any router configuration. Finally, the SSR-Controller has a performance monitor module that periodically measures statistics such as processor usage, network throughput, and free memory using hooks provided by XEN. Feedback from this module can be used by the SSR-Controller to wakeup some hosts and disable their corresponding VM images in case the load

on the SleepServer exceeds capacity. In case of multiple SleepServers on the same subnet, individual SSR-Controllers communicate with each other to provide load balancing and redundancy.

Although we have implemented SleepServer on a separate server machine, it is feasible to have a scenario where the SleepServer functionality can be supported on enterprise PCs themselves. A subset of enterprise PCs can run a hypervisor and the SSR-controller and can host the images of other PCs that are asleep thus proxying for them. However, there may be other implications of this approach that we have not fully evaluated such as maintaining security, resource allocation and isolation.

4.3 Host VM Image Image

The host images running on the SleepServer are XEN VMs based on the standard x86 architecture executing a stripped down version of Ubuntu Linux. After installation the VMs take up less than 300MB out of their initially allocated 1GB disk image. Given that only the essential services are run inside the VM, our initial memory allocation of 64MB is more that sufficient with most of the memory free (>40MB) after boot up.

Each VM is configured to have several software modules to support SleepServer operation, as illustrated in Figure 4. Each VM has a full TCP/IP stack and can therefore respond on behalf of the host to packets such as ICMP echo-requests or ARP queries. The VMs have a firewall based on the `iptables` package which is configured to be identical to the host firewall by the SSR-Controller. The 'Packet-Analyzer' (PA) module is used to support stateless applications such as incoming RDP or SSH requests. The PA module is based on the BSD raw socket interface and is used to parse incoming packets to look for matches on one or more fields of packet headers, such as incoming requests on particular TCP and UDP ports. In case the incoming packet matches one of the application ports, the firewall is configured to not send a response to the initial request. Instead the PA sends a message to the SSR-Controller and disables the network interface of the VM. Upon receipt of this message, the SSR-Controller uses the wakeup module to send a wakeup packet to the host as described earlier that relies on retries inherent in TCP. When the host resumes, it receives one of the retransmits and the session can be established. For stateful applications we have implemented application stubs similar to those proposed in Somniloquy [3]. We currently support several application stubs namely a multi-protocol instant messaging stub, a background web download stub and a BitTorrent stub that allows participation in P2P networks. Given the x86 compatible architecture of the VMs, and the availability of standard libraries and tools, implementing these stubs is significantly easier than on the specialized Somniloquy hardware device.

In a way, supporting a large number of stateful applications may be considered a barrier to deployment since each application requires its own corresponding stub. In our experience with deploying SleepServer, we did not observe this to be an issue, especially in enterprise settings, for several reasons. First, a significant portion of users can be supported without requiring stubs, as long as seamless connectivity (responding to ARPs, ICMP) and user selectable wakeup on incoming connections is handled (e.g. SSH, RDP, SMB, backups and updates). This observation is in fact similar to the findings of previous measurement based studies [5, 20] in this space. The rest of the users in our deployment requested support for a relatively small number of common services (and hence stubs). The "fall-back" position of waking up the PC provides a fail-safe default for those applications or protocols that we do not yet proxy. Indeed, additional stubs will only improve upon the energy results we report.

4.4 Discussion

An emerging use model in enterprising computing is based around Virtualized Desktops environments. The common scenario is when all desktops reside on a centralized server and users utilize thin clients to connect over the network to their desktops. In this setting we believe the lightweight proxying functionality that SleepServer offers can potentially increase the density of inactive desktops, thus improving scalability. Another scenario is based on the assumption that each Desktop PC runs a Hypervisor itself (e.g. XEN or VMware) and the actual users 'virtual desktop' runs as a VM on top of the VMM/hypervisor. When the user is in front of his actual PC, his virtual desktop runs on the local VMM and when the user steps away or logs out the entire VM can be migrated to a central server, or pool of servers. The advantage of this architecture over SleepServer is that no application stubs are needed since entire VMs are migrated. However, a potential drawback of this approach is its limited scalability since the amount of state that needs to be transferred for each virtual desktop can be quite large. For example, the memory footprint alone may be up to several gigabytes based on the hardware configuration of the host PC and the entire OS state, including all applications. is transferred and has to be kept running even if the user only wants a small subset of the functionality when they are away. Furthermore, the local persistent storage may also need to be migrated. Despite using techniques like memory ballooning and deduplicating memory, the scalability of a virtual desktop based infrastructure will most likely be significantly limited than that of SleepServer which uses very lightweight

	Machine Type	Year	OS	Average Power in S3	Average Power when on (idle)	Time to Resume from S3 (network)
1	Dimension 4500	2002	WinXP	2.5 Watts	75 Watts	29 (+/- 4.1) seconds
2	Dimension 4500	2002	WinXP	2.4 Watts	61 Watts	28 (+/- 1.6) seconds
4	Dimension 4600	2004	WinXP	4.4 Watts	76 Watts	29 (+/- 3.0) seconds
4b	(Same as above - Dual Boot)	2004	Ubuntu	4.6 Watts	74 Watts	12 (+/- 1.8) seconds
5	Dimension 4700	2005	WinXP	2.2 Watts	111 Watts	30 (+/- 10.0s) seconds
6	Optiplex SX260 Small Form Factor	2004	Ubuntu	5.1 Watts	67 Watts	10 (+/- 7.44) seconds
	(Desktop + SSH/CVS Server)					
7	Optiplex GX280 Small Form Factor	2005	WinXP	3 Watts	86 Watts	25 (+/- 5.3) seconds
8	Optiplex 755	2007	Ubuntu	2.8 Watts	84 Watts	14 (+/- 2) seconds
	(Desktop, SSH + file server)					
9	Optiplex 745	2007	Vista	3.3 Watts	107 Watts	8 (+/- 1.4) seconds
10	DELL XPS 720	2008	Win XP	4.2 Watts	314 Watts	9 (+/- 7.7) seconds
	(Drives LCD Display + Webserver)					
11	Optiplex 960 Small Form Factor	2009	Win 7	2.3 Watts	45 Watts	12 (+/- 5) seconds

Table 1: *Power consumption for an example set of PCs in our deployment. Resume from S3 times are much better for newer machines and operating systems. Base power consumption of newer PCs still remains high and power consumed in S3 ranges from 1/20 to 1/75 of that in idle mode.*

VM images which can be as low as 32MB in footprint. Going forward, we do believe that Virtual Desktop based solutions and the significantly lightweight proxying approach offered by SleepServer are synergistic and both may be useful based on specific use cases. For example, a SleepServer can potentially host full virtual desktops when particular stubs may not be available. We also believe that any investments that application vendors make into building stubs for their applications will be useful for both a lightweight VM based approach taken by SleepServers, as well as potential hardware solutions that add proxying functionality to network interface hardware[3, 16].

5 Evaluation

We first present micro benchmarks highlighting our experience with deploying SleepServers to various hosts in our department. We then evaluate the SleepServer itself, benchmarking its power consumption under various loads and measuring the latencies for management tasks such as creating, starting and shutting down new host images. We also present experimental data about the scalability of SleepServers demonstrating that we can easily scale to serve several hundred hosts on a single SleepServer machine. Finally, we present data that shows the energy savings of the various host PCs in our deployment.

5.1 Micro Benchmarks

We have deployed SleepServer on a variety of host PCs in our department building. In total we have over thirty desktop PC users including a couple of laptop users participating in our SleepServer deployment. The users range from faculty and students to full time staff workers to give us a mix of use-scenarios. Also, the mix of machines range from PCs that are well over 7 years old to those that are fairly new. The operating systems running on these PC range from Linux (Ubuntu) to all versions of Windows, including numerous Windows XP machines.

Table 1 shows the distribution of some representative PCs that are part of our SleepServer deployment. Our goal in benchmarking these systems was to see whether we could observe any trends in the design of PCs and operating systems. We benchmarked these systems based on power consumption in various states of operation, and the latency of these systems when they resume from sleep. We only show the latency measurements to resume from sleep, since latency to go to sleep is less important from a usability standpoint. We have instrumented all the machines in our deployment to provide real time energy measurements using a commercial energy meter from WattsUP devices[1]. We have also made this energy data available to SleepServer users to view over the web using an 'Energy-Dashboard' interface that we have designed [4]. In addition to viewing their power usage in real time, users can also look at long term trends such as comparing their usage over different time periods.

Our instrumentation of the thirty desktop computers in our SleepServer deployment using energy meters gives us long term power use data, allowing us to measure and quantify the impact of using SleepServers under different usage scenarios. We observed that most users in our deployment did not put their machines to sleep before

[1]www.wattsupmeters.com

they started to use SleepServers, as measured by over five months of power usage data by these machines.

Table 1 reports the power consumption and latency values for an example set of SleepServer PCs. We do not include the power consumed by LCD displays connected to these PC, since most of them are configured to go into sleep modes on inactivity. Several interesting observations can be made from the table. First, the power consumption in sleep (S3) mode for most of the PCs is significantly less than when they are in idle mode. This is even true across operating systems (line 4 and 4a for the same PC in the table). Second, the power consumption of PCs has not come down significantly during the last 7-8 years, as idle power for desktops remains around 80 Watts for even new PCs. Third, the latency to resume from sleep varies significantly across platforms. We measure the latency to resume by measuring the time from a wakeup event, such as a key press on the keyboard, to the time it takes for the network stack on the host PC to respond to an incoming ICMP packet. Although the display and logon screens on the host may come up earlier, we believe measuring the latency for a network response is a better metric to use. Table 1 shows that in some cases resume latencies are up to 30-40s (line 5), with a large standard deviation in time to resume. We also notice that the resume time on different operating systems (line 4 and 4a) on the same hardware platform are significantly different. We believe this is mostly due to the different applications, devices and drivers that are installed on PCs over time and can cause delays in startup. Importantly, as we can see from the table, resume times are getting significantly better as we move to more recent hardware (2007 and newer) and modern operating systems.

5.2 Scalability of SleepServer

The hardware and software configuration of our SleepServer prototype was presented earlier in Section 4. We measured the power consumption of our prototype under various operating conditions using a WattsUP device. We also measured the latency to create a new SleepServer image for a particular host, and the time to start up an existing VM and shut it down. These latencies are important to consider for dynamically creating new VMs when new hosts are added to a SleepServer.

The latency and the power consumption values are shown in Table 2. The latency to create a new host image from scratch is on the order of two minutes. This includes creating the image, installing the SleepServer supporting software, configuring the SleepServer controller and updating all packages and security updates. To reduce this latency, the SleepServer allows creation of a pool of VMs, which can be updated with the network

	SleepServer function	Time (seconds)
1	Creating a new host image	120s (+/- 10)
2	Starting up a host image	11s (+/- 1)
3	Shutting down a new host image	12s (+/- 1)
	Sleep-Server - State	**Power (Watts)**
4	Idle State, no host images running	213 W
5	Hosting 200 *idle* host images	221 W
6	Download + Write to Disk	255 W
7	CPU benchmark, (100% CPU util.)	308 W

Table 2: *Benchmarking the Sleep-Server: Latency and Power Measurements*

configuration of a host. The time to start up an existing VM and shut it down is around ten seconds. To reduce the startup latency even more we have enabled only the essential services in the VMs. This latency is important, since it means that given the transition times presented earlier in Table 1, it is possible to *dynamically* start up VMs and have them activated by the time the host finishes its transition to sleep. Alternatively, the host VMs can be started up if memory and CPU on the SleepServer are not a constraint. Finally, the time taken for our prototype SleepServer machine to boot up from a powered off state, to recreate state information from its logs, and to start up the VM images is on the order of a few minutes.

Next, we tested the scalability of our SleepServer prototype by instantiating a large number of VMs on it and measuring the effect on the processor and the memory utilization and impact on I/O performance. Since we allocate 64MB of memory to each VM, that gives an upper bound of approximately 500 VMs executing simultaneously for the 32GB of main memory in our SleepServer prototype. Unfortunately due to some limitations in XEN and the Linux kernel, we were unable to scale beyond 200VMs. The limitations relate to the low number of statically defined software interrupts in the XEN kernel, as well as the number of block devices (disks) supported. We have reported these limitations and the fix should be released in an upcoming update.

Figure 5a shows how increasing the number of VMs impacts the overall CPU and the memory utilization of the SleepServer. The processor utilization increases linearly and remains low (20%) even at 200 VMs (idle), giving almost 80% idle time for the CPU. The low CPU utilization is as expected, since most of the idle VMs are in a blocked state waiting for I/O (e.g. network packets) requests. The memory utilization also increases linearly as we increase the number of VMs, since each VM uses an additional 64 MB. Next we benchmark the performance of these VMs under I/O load, by setting up an experiment where a number of VMs download data from a fast local webserver using a web download stub. As

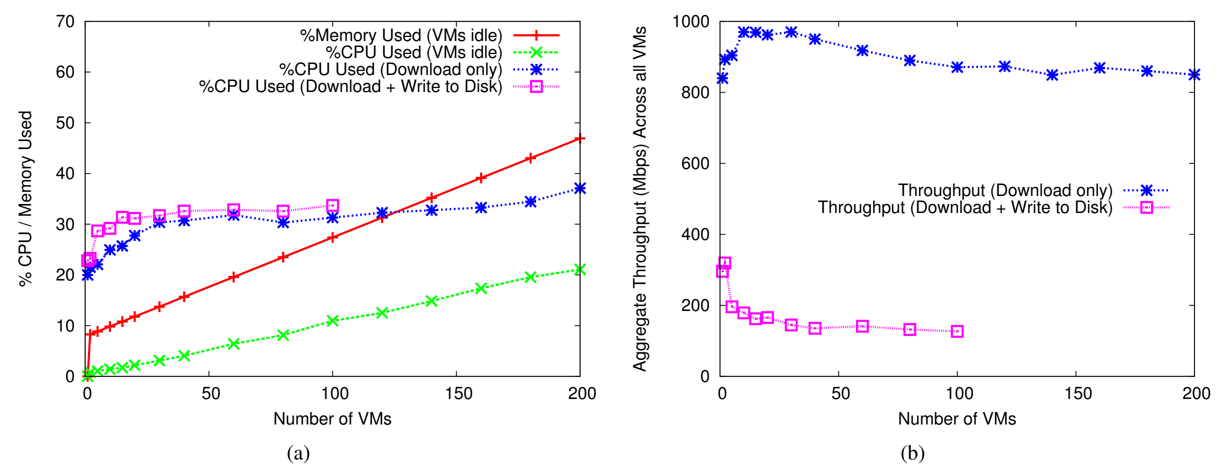


Figure 5: *Effect of scaling the number of VMs. The graph on the left (a) shows the memory and the CPU utilization as we increase the number of VMs. The amount of memory used under additional network traffic by the individual VMs does not change and is therefore not shown. The graph on the right (b) shows the total aggregate throughout observed by all the VMs as we increase the number of downloads.*

we increase the number of VMs simultaneously running this benchmark, we measure the CPU and memory usage and the aggregate throughput observed by the VMs. We report these figures for two cases: when the download is not saved to disk marked as ‘Download Only’, and when the VMs save the downloaded data to their local storage marked as ‘Download + Write’. When the VMs are not saving the data to disk, the aggregate network throughput is shared evenly between all VMs, and the downloads almost saturate the 1Gbit link (>800Mbps for 200VMs). The CPU utilization increases to 40% even at 200 simultaneous downloads. However, when the VMs are writing to disk, CPU utilization rises to about 35%, while the download throughout reduces to about 136Mbps (Fig 5b) for 50 simultaneous downloads and to 126Mbps for 100 downloads. This can be explained by disk seek times, caused by each VM writing to its image, starting to dominate as the number of VMs increase, thus limiting performance. We did not measure ’Download Write’ performance beyond 100VMs since we started to observe disk driver timeouts for some of the VMs. Using faster disk drives or striping the VMs across separate local hard drives on the SleepServer, or by using a network storage element should expectedly improve performance. As discussed in Section 4 earlier, the SSR-Controller can detect this condition and choose to wake a few of the hosts to alleviate any I/O bottlenecks. We measured the average ICMP latency from a local machine to the VMs on the SleepServer, as a measure of network responsiveness of the VMs under load. The round-trip latency was under 5ms under all conditions.

The primary goal of SleepServer is to enable users to put their PCs to sleep to save energy, while maintaining their availability to both network and application level events. Using the power consumption logs captured by the WattsUP meters, we can calculate the energy consumed by the various PCs over different periods of time and use that to calculate the energy savings. The energy savings for users is also dependent on how often users actively put their machines to sleep. As an experiment, we first let users use SleepServer in a mode where they were responsible for putting their machines to sleep manually. For the next week we modified the standard power management settings for some users such that after one hour of idleness, as detected by the power management functions of the host OS, the PC would automatically go to sleep. Note that for both these cases, the users were aware that they would be able to use the SleepServer functionality to access their machine and maintain connectivity when their computers were in sleep mode.

Figure 6 shows the power consumption trace for a typical user of our system drawn from the group of thirty users. This figure compares the power consumption of the user’s PC over a 2 week period, first without SleepServer (August 31st - September 13th) and then when the user started to utilize SleepServer (September 14th - September 27th). Additionally, for the first week of deployment (Sept 14th - Sept 20th) the users were asked to put the machine to sleep manually when it was not in use, while for the second week (Sept 21st - 27th) the one hour idle-timeout was instituted. For the first week the energy consumption of this user dropped by 30% as seen by the frequent transitions to sleep. There were however several cases during the first week when the users forgot to put their machines to sleep despite the fact that they were not actively using the PC (e.g. Sept

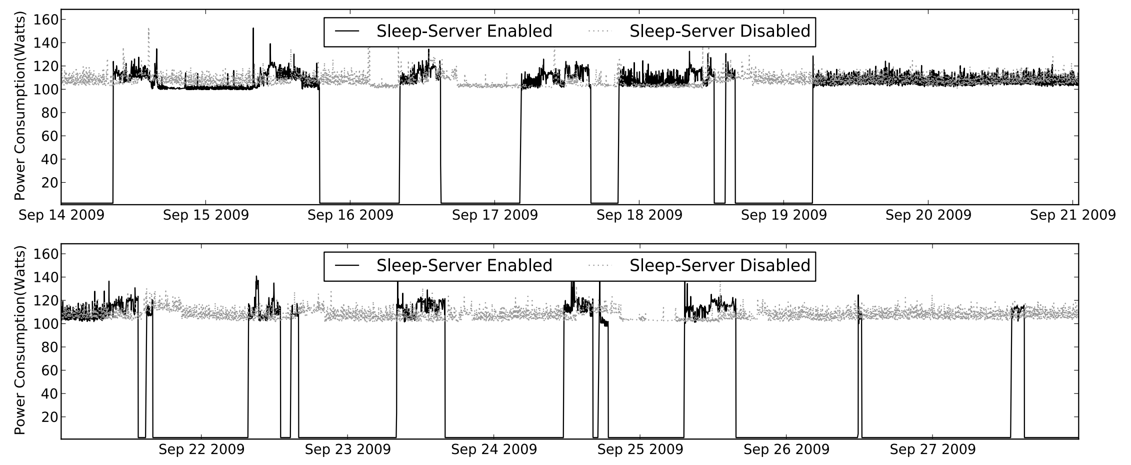


Figure 6: *Comparing the Power Consumption for a Desktop PC with and without Sleep-Servers. For the first two weeks from August 31st - Sept 13th the user was not using SleepServer, while from Sept 14th to September 28th, SleepServer operation was enabled. Additionally, from Sept 22nd onwards the PC was set to automatically go to sleep within one hour of idleness.*

15th, Sept 20th and 21st). In Week 2, when we instituted the one hour timeout policy, there were more transitions to sleep (Sept 22nd, Sept 24th), even during the day. The end result was that the user saved an additional 54% energy between Sept 21st - 27th over the previous week, giving a total enegy savings of 68% over the period where the PC was always on. We also notice that the user logged in to his PC remotely during the weekend (September 27th and 28th), and that the PC went back to sleep afterwards.

5.3 Energy Savings Using SleepServer

Of course, the energy savings for a particular user or a PC are based on its usage scenario. Graduate students in our department tend to stay longer, while most staff and faculty have relatively fixed hours. A significant fraction of people do however connect to their PCs remotely, and in some cases even run services like a web server or a CVS repository on their machines which would normally preclude them from putting their machines to sleep. Using SleepServer our entire deployed set of more than thirty users were able to put their machines to sleep. In Figure 7 we have plotted a representative set of eight host PCs for two weeks in September 2009. To simplify the chart, we have plotted a step function denoting the state of these PCs rather than absolute power consumption values. The times when the host is active (and its image on the SleepServer is disabled) is marked by an ’A’, while ’S’ marks the times when the PC is asleep (and its image on the SleepServer is enabled). The hosts are ordered from top to bottom in terms of energy savings, with PC1 seeing the most savings and PC8 seeing the least.

There are several important observations from Figure 7. First, we can clearly see the advantages of instituting the one hour idle timeout for certain users. Users of PC2 and PC3 forget to put their machines to sleep and as a result their PCs remained on through the weekend of Sept 12th/13th (marked by a ‘1’ in the chart). When the automatic timeouts were instituted, most of the PCs remained asleep for longer periods of time including over the weekend of September 19th/20th (marked by a ‘3’ in the chart). Second, while there were some trends in terms of machines being turned on in the morning when the users came in to work, the distribution of when the machines are on or sleeping using SleepServers is quite varied over the week. Users of PC4 and PC8 for example log in to their PCs to work over the weekend (marked by a ‘2’ in the chart). This points to the fact that a simple scheduled policy of waking up PCs at pre-determined work times does not suffice. By mining the SleepServer controller logs we can also determine what caused particular PCs to wakeup. PC1 for example runs a Web server; any request to access the website therefore causes the SleepServer to wake up the machine. After a configured period of inactivity PC1 goes back to sleep causing frequent state changes. It is important to note that a major fraction of the users in our deployment were running one or more application that otherwise would have required the user to keep their machine powered on. During the course of our study, for example, our measurements show that 22 out of the total 30 machines needed to be woken up. Furthermore, despite the limited number of stubs we currently support, 6 out of 30 users utilized one or more stubs during our study. Of course the particular stubs that are required may depend on the enterprise environment, and we expect to gain more experience as

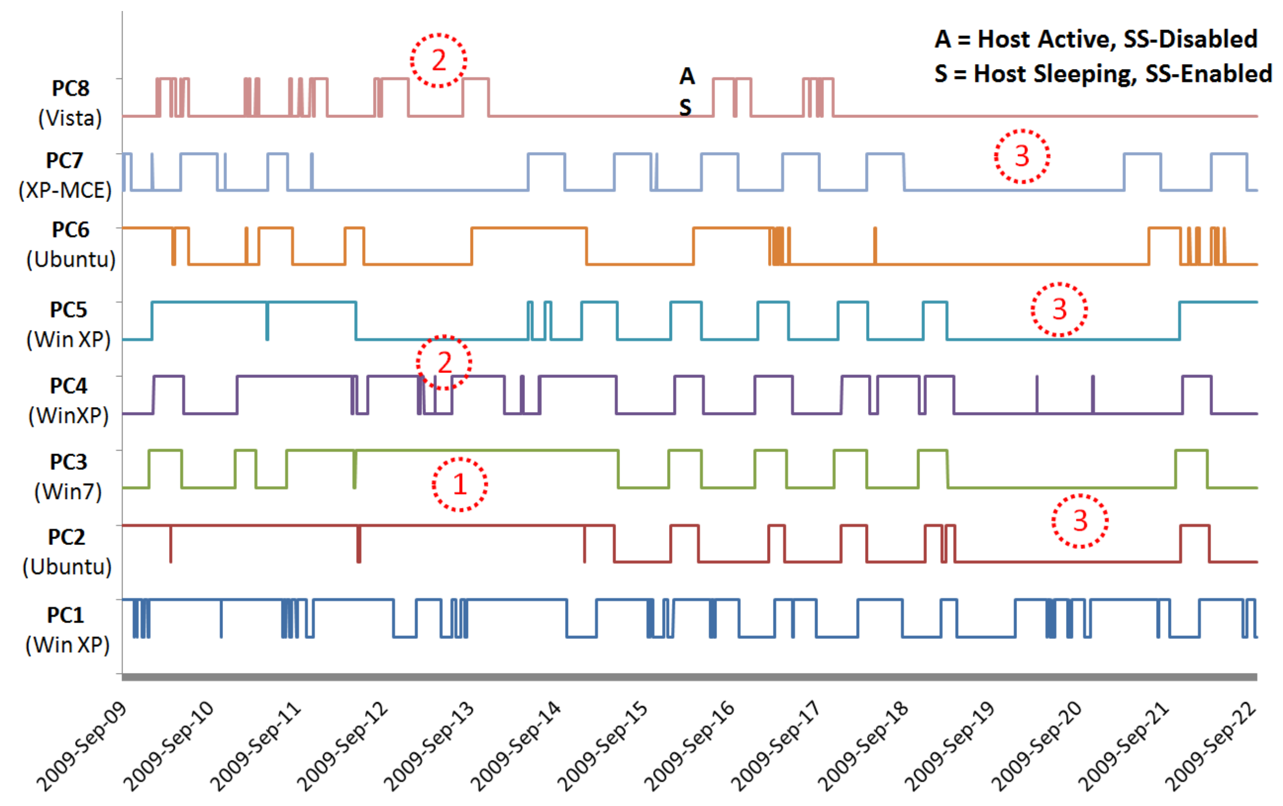


Figure 7: *Showing eight different hosts on using SleepServer over a two week period. For each host the graph shows the times when the host was on and its image on the SleepServer was disabled (denoted by A) and when the host as asleep and the SleepServer was proxying for it (denoted by S).*

we deploy SleepServers further. However, we do believe that it is important to support the capability of handling stateful applications in the SleepServer architecture for widespread adoption. The energy savings for the example set of 8 PCs shown in Figure 7 is significant, ranging from 27% (PC1) to 81% (PC8) for this two week period. The measured energy savings across all machines in our deployment for the month of September range from 27% to 86%, with an average savings of 60%.

6 Conclusion

In this paper we have presented SleepServer, a software-only implementation of proxy architecture that allows end hosts to utilize low power sleep modes frequently and opportunistically to save energy, without sacrificing network connectivity or availability. Within enterprise networks, a SleepServer machine can maintain network presence on behalf of a host while its sleeping by responding on behalf of the host seamlessly and waking it only when required. SleepServers are easily deployable since they require no changes to existing hardware, software or networking infrastructure and can be supported entirely using a simple software agent on the end hosts. We demonstrate that SleepServer is portable across a range of operating systems and hardware platforms and show how our prototype implementation can scale to support hundreds of client hosts on a single commodity server.

SleepServer is both practical, easy to deploy and very scalable. A large number of clients (and thus VMs) that are doing intensive disk activity might limit scalability due to heavy disk I/O. However, a case can be made to limit or avoid putting such machines to sleep. Instrumenting thirty heterogeneous desktop users, we show energy savings ranging from 27% to 86% with an average savings of 60%. Extrapolating from these results and assuming an average idle power consumption of 93Watts per desktop (from Table 1), and a use factor of 40% (machines are asleep for 60% of the time), we expect to reduce the baseline power use of the CSE building from 320KW to 245KW during nights and weekends. At current California energy prices of 9 cents per KW-Hr, this translates to over US$ 35,000 in annual cost savings alone, easily paying for the cost of a Sleep-Server within a few months.

7 Acknowledgements

We would like to acknowledge Thomas Weng for his feedback on the paper and his invaluable help with setting up the energy vizualization framework, the Energy Dashboard. We also wish to thank the anonymous reviewers of our paper for their detailed feedback. Finally, we are grateful to our shepherd, Orran Krieger for guiding us towards the final version of the paper. This work is in part supported by the NSF CNS-0932360 grant and Multiscale Systems Center (MuSyc) under the Focus Center Research Program (FCRP) supported by DARPA/MARCO.

References

[1] ACPI. Advanced Configuration and Power Interface Specification, Revision 3.0b, 2006.

[2] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless Wakeups Revisited: Energy Management for VoIP over Wi-Fi Smartphones. In *MobiSys '07: Proceedings of the 5th International conference on Mobile Systems, Applications and Services*, 2007.

[3] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, 2009.

[4] Y. Agarwal, T. Weng, and R. Gupta. The Energy Dashboard: Improving the Visibility of Energy Consumption at a Campus-Wide Scale. In *in BuildSys '09*.

[5] M. Allman, K. Christensen, B. Nordman, and V. Paxon. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *6th ACM Workshop on Hot Topics in Networks (HotNets)*.

[6] Apple. Wake-on-Demand. `http://support.apple.com/kb/HT3774`.

[7] 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. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[8] J. Chase, D. C. Anderson, P. Thakkar, A. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of SOSP '01*, 2001.

[9] DOE. Buildings Energy Data Book, Department of Energy, March 2009. `http://buildingsdatabook.eren.doe.gov/`.

[10] DOE. CEC End-Use Survey, CEC-400-2006-005, March 2006. `http://www.energy.ca.gov/ceus/`.

[11] K. Flautner, S. K. Reinhardt, and T. N. Mudge. Automatic Performance Setting for Dynamic Voltage Scaling. In *Proceedings of MobiCom '01*, 2001.

[12] J. Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, 2004.

[13] D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, 2008.

[14] M. Gupta and S. Singh. Greening of the Internet. In *SIGCOMM '03: Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003.

[15] Intel. Intel Remote Wake Technology. `http://www.intel.com/support/chipsets/rwt/`.

[16] M. Jimeno, K. Christensen, and B. Nordman. A Network Connection Proxy to Enable Hosts to Sleep and Save Energy. In *IEEE IPCCC '08*.

[17] P. Lieberman. Wake-on-LAN technology. `http://www.liebsoft.com/index.cfm/whitepapers/Wake_On_LAN`.

[18] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of ASPLOS '09*. ACM New York, NY, USA, 2009.

[19] R. Nathuji and K. Schwan. Virtualpower: Coordinated Power Management in Virtualized Enterprise Systems. *ACM SIGOPS Operating Systems Review*, 41(6):265–278, 2007.

[20] S. Nedevschi, J. Chandrashekar, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: reducing energy waste in networked systems. In *Proceedings of USENIX NSDI '09*.

[21] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proceedings of USENIX NSDI '08*.

[22] 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.

[23] A. Qureshi, H. Balakrishnan, J. Guttag, B. Maggs, and R. Weber. Cutting the Electric Bill for Internet-Scale Systems. In *SIGCOMM*, 2009.

[24] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *MobiCom '02: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 160–171, New York, NY, USA, 2002. ACM Press.

[25] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical Power Management for Mobile Devices. In *MobiSys '05: Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services*, 2005.

An Extensible Technique for High-Precision Testing of Recovery Code

Paul D. Marinescu, Radu Banabic, George Candea
School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Thorough testing of software systems requires ways to productively employ fault injection. We describe a technique for automatically identifying the errors exposed by shared libraries, finding good injection targets in program binaries, and producing corresponding injection scenarios. We present a framework for writing precise custom triggers that inject the desired faults—in the form of error return codes and corresponding side effects—at the boundary between shared libraries and applications.

We incorporated these ideas in the LFI tool chain [18]. With no developer assistance and no access to source code, this new version of LFI found 11 serious, previously unreported bugs in the BIND name server, the Git version control system, the MySQL database server, and the PBFT replication system. LFI achieved entirely automatically 35%-60% improvement in recovery-code coverage, without requiring any new tests. LFI can be downloaded from `http://lfi.epfl.ch`.

1 Introduction

Most software interacts with its environment through libraries, and most environmental events, including failures, are exposed to applications through the APIs of these libraries. Shared libraries, in particular, are widely used, as they encapsulate frequently used functionality. General-purpose applications frequently link to tens or even hundreds of shared libraries [18].

As a result, the reliable functioning of programs is tightly coupled to how well they handle the returns from shared libraries. While most of the APIs are well documented, they can be quite complex, and they differ from platform to platform in subtle enough ways to not be noticed during porting, but in sufficiently important ways to cause problems (e.g., `errno` values for the same `libc` call can vary between Linux, Solaris, and Mac OS X). Such corner cases are easy to miss and can lead to crashes or more subtle errors. For example, a `read()` call may not be retried after receiving an `EINTR` return code, causing the application to miss some data in its input stream.

These bugs are hard to find through input testing alone, because they are triggered by low-probability events that are typically input-independent and occur below the library layer. Yet there must be a way to ensure that programs with high reliability requirements, such as servers or embedded applications, use these libraries consistently with the libraries' true behavior. In particular, it is essential to verify that such software can correctly handle faults at or below the library layer, faults that manifest as errors returned through the shared libraries' interface.

The program/library boundary is an appealing location for injecting faults. First, it provides a well defined API where realistic faults can be injected. Second, most error recovery code can be exercised directly or indirectly via library-level fault injection. However, one must intelligently restrict the number of fault injections—an exhaustive injection campaign is infeasible, while a random one is unlikely to find bugs in a reasonable amount of time. One way to achieve this restriction is to target fault injection to precisely the program conditions that are of interest for testing, and none other.

This paper introduces a mechanism for high-precision fault injection. We extend our LFI library-level fault injector [18] with three new techniques: First, a *fault injection triggering* mechanism that allows testers to specify conditions under which a given call to a given library should be caused to fail. Triggers take the form of predicates on program state—global and local variables, call stack, etc.—which enable arbitrarily precise control over the fault injection process. Second, we present a *call site analysis* technique that aids in constructing these triggers by automatically identifying potentially buggy recovery code in program binaries, along with information on the ways in which the library calls at those sites can fail. The LFI analyzer automatically produces injection scenarios to exercise these code areas. Finally, we present an expanded *fault injection language*, which allows testers to devise sophisticated fault injection scenarios, as well as combine base triggers to form new triggers, without having to write any new code.

The rest of the paper provides an overview of LFI (§2), describes the triggering mechanism (§3), fault injection language (§4), and call site analysis (§5), after which we present details of our implementation (§6), evaluate our system (§7), survey related work (§8), and conclude (§9).

2 System Overview

We combine a library-level fault injector with three new elements that turn it into a high-precision testing tool: fault injection triggers, a fault injection language, and call site analysis. A developer would employ call site analysis to automatically identify good targets for testing, then potentially refine the generated test scenarios (written in the fault injection language), and finally provide all these to the fault injector. Below, we provide a brief overview of the injector, along with the three main contributions of this paper.

In order to make testing based on fault injection more accessible to programmers, we developed LFI [18], a tool for injecting faults at the boundary between programs and the shared libraries they use. LFI generates shim libraries to intercept library calls; based on predicates generated from user-provided configuration, they decide to either pass control to the original function or return an error value to the calling program (Figure 1).

The state of the art in testing software by injecting high-level faults consists largely of hand-coding the tests inside the product itself. For example, the MySQL server code has occasional custom code that returns specific errors when activated via a compile-time debug directives. In contrast to this approach, LFI decouples the testing from the target system's code, thus enabling automation and reuse of fault injection tests across many systems.

Using LFI involves two steps: First, a fault injection scenario is developed either by hand or using one of the automated techniques described later on. Second, the scenario is provided to the LFI controller, which conducts a suite of tests in which the described errors are introduced in the library API. The output of these tests can be used to diagnose and fix the identified bugs.

Underneath the covers, LFI uses the fault injection scenario to synthesize custom interposition libraries. The synthetic libraries have the same API as the original ones, but underneath the API they encode the fault injection logic. These libraries are shimmed between the program being tested and the original library(ies); multiple such synthetic libraries can coexist simultaneously. They intercept calls of interest, made from the program to the shared libraries, and return erroneous results that simulate faults in the libraries and the environment, as required by the scenario. The shimming of the generated libraries is system-specific: on Linux and Solaris, LFI uses the `LD_PRELOAD` mechanism to communicate

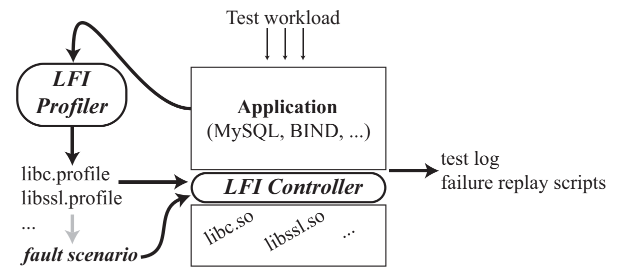


Figure 1: Architecture of the LFI fault injector.

with the dynamic linker, while on Windows it uses the Microsoft Detours framework [14].

The LFI controller coordinates the entire testing process. It is responsible for interpreting the injection scenario, generating the corresponding interception stubs, and combining them with runtime code and triggers to synthesize a new library. Once the stubs are generated and installed, the controller invokes a developer-provided script that starts the program under test, exercises it with the desired workload, and monitors its behavior to determine whether it terminates normally or with an error exit code. This information is collected in a log used by developers to diagnose and fix the program.

The LFI log records each error injection, the injected side effects (e.g., `errno`), and the events that triggered that injection (e.g., call count, stack trace). This information can be used to match injections to observed program behavior, as well as to refine the fault scenario. This also helps pinpointing and fixing the bug that caused the failure. Third party systems, like R2 [13], can be used to replay deterministically all program failures of interest.

In order to help with the generation of fault scenarios, LFI provides an automated *library profiler*, which operates directly on the binaries of the shared libraries. It performs two tasks: First, using static analysis of the binaries, it infers the return codes of the functions exported by a library (e.g., it determines that `read()` in libc can return -1, 0, or a positive number). Second, it infers side effects—besides error return values, library functions may communicate to callers additional information regarding the encountered error, via channels such as output parameters, global variables, or thread local storage (TLS) variables. For example, the profiler finds that, when returning -1, `read()` could also set the TLS variable `errno` to `EAGAIN`, `EBADF`, `EINTR`, etc. The results of these analyses are output in an XML file representing the so-called *fault profile* of the target library.

The present paper shows how we extended LFI with new techniques for writing and running sophisticated tests with little effort. Some of the key questions we address include: How to specify exactly at which point in a program's execution to inject errors? When testing a large program, how to decide where to inject faults? Figure 2 illustrates the three new elements: a high-precision triggering mechanism, along with an expanded fault injection language and a call site analyzer.

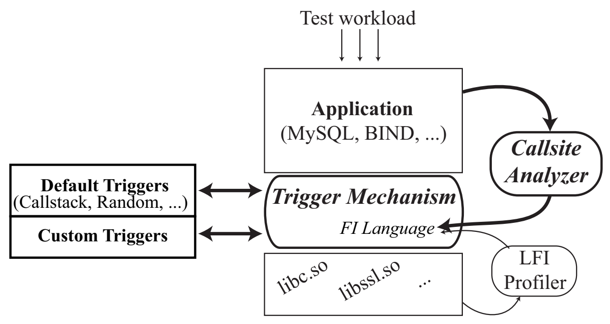


Figure 2: Architecture of the new injection framework.

A fault injection trigger is a way to specify which fault ("what") to inject at which point in a program's execution ("when") and in which call to the target library ("where"). This trigger is a predicate that evaluates to true when a fault should be injected and false otherwise. The new LFI includes default stock triggers as well as an API for writing custom triggers.

The fault injection language glues together triggers, library functions, and their fault profiles into complete fault injection scenarios. Every function which appears in a scenario is intercepted by the LFI runtime, and the associated triggers are called to decide (based on various conditions) whether to inject a fault from the profile.

The call site analyzer uses a heuristic method to check whether all error return values are checked by the callers of library functions. In other words, it searches the target program for "interesting" places to inject faults. For each identified call site, it uses dataflow analysis to determine against which error code values the return is checked. An unchecked error value indicates a potential bug, to be verified through testing. The analyzer runs on the binary, so source code of the target program is not required.

Next we describe the fault injection triggers (§3), fault injection language (§4), and call site analysis (§5).

3 Fault Injection Triggers

Triggers are invoked by the LFI runtime to decide whether an intercepted library function should fail or not. A trigger can inspect any part of system state to make its decision. Triggers offer high precision and flexibility to testers in choosing the exact conditions under which a fault is to be injected. Testers can use stock triggers with specific parameter values, they can customize existing trigger code, or write new triggers from scratch.

3.1 Trigger Interface

Triggers are pluggable modules, written as C++ classes that implement the following `Trigger` interface:

```
class Trigger {
   virtual void Init(xmlNodePtr initData) {}
   virtual bool Eval(const string&
                          libFuncName, ...) = 0;
}
```

To add a new trigger to the framework, one writes a class derived from this abstract base class and places the corresponding source files in an LFI-specific location. The new trigger can be referenced directly by class name from any injection scenario provided to LFI. We used a variant of the Registry design pattern and the Standard Template Library to implement this behavior transparently for trigger writers.

The boilerplate code needed for a trigger is minimal—usually less than 100 lines of code are needed to write a useful custom trigger. One could forgo the interface and implement triggers inside the LFI runtime, as we did in a first prototype, but then they become hard to extend and require intimate knowledge of LFI internals.

The `Init` method is optional and its default implementation is empty. It is called by the runtime after a trigger instance is created and before its `Eval` function is called for the first time. The main purpose of the `Init` function is to provide support for trigger parametrization, as we will show in §4.1. The trigger's parameters are provided as an `XMLNodePtr` object, which can be processed with a standard XML library.

The `Eval` method is where the main trigger logic resides. It is called every time a function (associated with an instance of this trigger by the injection scenario) is intercepted. Its return value indicates to the runtime whether to inject a fault or not. Since `Eval` can be called quite frequently, its code must be efficient.

`Eval` is a variadic function in order to be capable of receiving the original arguments of any intercepted library call. Its first argument indicates the name of the intercepted function; based on this name, the trigger decides how many actual arguments to expect and what their meaning is. The number of arguments must be explicitly specified in the injection scenario; since LFI does not look at source code or documentation, it cannot automatically infer the number of arguments to pass. It is possible, though, to extend LFI with LibTrac's heuristic technique for inferring function arguments [5].

In addition to the arguments passed to `Eval`, a trigger can directly obtain any other information normally accessible to a program. For example, it can use the GNU libc `backtrace()` function to inspect the call stack and determine whether the intercepted library call was made by a program, by a function in the intercepted library, or by some other shared library.

Below we illustrate trigger construction with a sketch of one of our custom triggers. It is used in an injection scenario where errors are to be injected in `read` whenever the corresponding file descriptor is a pipe, the number of bytes to be read is between 1 KB and 4 KB, and the calling thread holds a POSIX mutex.

```
#include "Trigger.h"

DECLARE_TRIGGER ( ReadPipe1K4KwithMutex )
{
private:
  static __thread int lockCount;
public:
  ReadPipe1K4KwithMutex() { }

  bool Eval(const string& libFuncName, ...) {
    pthread_t self = pthread_self();
    if (libFuncName ==
          "pthread_mutex_lock") {
      ++lockCount;
    } else if (libFuncName ==
          "pthread_mutex_unlock") {
      --lockCount;
    } else if (libFuncName ==
          "read") {
      if (lockCount > 0) {
        va_list ap;
        int fd;
        size_t size;
        struct stat st;
        va_start(ap, libFuncName);
        fd = va_arg(ap, int);
        va_arg(ap, void*);
        size = va_arg(ap, size_t);
        va_end(ap);
        fstat(fd, &st);
        return (S_ISFIFO(st.st_mode) &&
          size >= 1024 && size <= 4096);
      }
    }
    return false;
  }
};
```

`Trigger.h` contains all the required definitions for the trigger mechanism, including the definition of the `Trigger` interface. `DECLARE_TRIGGER` is a macro that simplifies the class declaration by automatically creating a properly derived class with the supplied name.

Triggers are not pure functions, but can also maintain state to inform their injection decisions. The `ReadPipe1K4KwithMutex` trigger, for instance, checks the name of the intercepted function and either updates the lock holding status for the current thread (in the case of a POSIX threads library call) or checks whether a fault should be injected in the case of the `read` function.

3.2 Stock Triggers

In addition to the `Trigger` interface, which allows testers to write their own custom triggers, LFI also provides a set of triggers that can be used out-of-the-box. We found this set to offer sufficient precision for most injection testing. Most of these triggers are generic, in that they can be used for any intercepted function and do not rely on the function's arguments. Stock triggers can easily be extended and specialized, as needed. LFI provides by default the following six triggers:

Call stack-based triggers allow injecting faults based on whether the current call stack (or part of it) matches a user-defined set of stack frames. By looking at the call stack, the trigger can learn from which particular location in the program the call is being made, from which program module (e.g., from Apache's SSL module), etc. User-provided stack frames can be identified by object file name, offsets within the binary, file name/line number pairs, or combinations thereof. The LFI call site analyzer automatically produces fault injection scenarios that use call stack-based triggers to inject faults in the locations where error checking is incomplete.

Program state-based triggers inject faults depending on whether a given relationship between program variables holds (e.g., *numConnections*==*maxConnections*). The stock trigger allows checking for equality between both local and global variables, but it can be easily extended with other operators. Later in the paper, we show how to specialize this trigger for data structures specific to the Apache Web server.

Call count-based triggers allow specifying that an injection should occur exactly on the *n*-th call to a function. Besides their obvious use, such triggers can also be used during debugging to replay observed failures in programs that are driven deterministically by interactions with the environment.

Singleton triggers inject a fault exactly once. This type of trigger is often combined with other triggers in a conjunction. For example, combined with a program state-based trigger, a singleton trigger can ensure that a fault is injected only the first time *numConnections*==*maxConnections* holds, but not afterwards. Trigger composition is described in more detail in §4.2.

Random triggers inject a fault with a configurable probability. These triggers can also be augmented with supplementary conditions, through composition.

Distributed triggers are used for testing distributed systems. A central controller receives information on intercepted calls (function name, arguments, and stack) and, based on a global view of the system, decides whether the remote trigger should fire or not. This allows setting up distributed failure scenarios, such as the ones we used for PBFT (see §7.1). In order to minimize runtime overhead, distributed triggers must be carefully composed with node-local triggers, so that the central controller is invoked solely when the injection decision can no longer be made locally.

In theory, one should write triggers that achieve perfect precision, i.e., they decide to inject a fault only in the specific situation targeted by the tester. However, in our experience, such high precision is not always ideal: it takes longer to write an ultra-precise trigger, and the induced runtime overhead can become non-negligible. In most cases, we favor an approach where triggers are *precise enough*, i.e., inject in all targeted situations and perhaps have a couple of false positives. In the context of testing, the extra unintended fault instances might even turn out to be useful, at little extra cost.

Care must be taken to not inject *unrealistic faults*, because these can result in wasted debugging time. For example, injecting an error in an I/O call made with a blocking file descriptor and setting `errno` to `EAGAIN` is arguably unnecessarily paranoid testing, given that `EAGAIN` should only occur on non-blocking file descriptors. In LFI, this exception could be handled by composing with a trigger that evaluates to true only when the file descriptor supplied to the I/O call is non-blocking (e.g., the trigger can check the file descriptor with `fcntl`).

4 Fault Injection Scenarios

An LFI test scenario is a declaration of triggers and conditions under which these triggers should be evaluated, i.e., it is a specification of what, when and where to inject. We use an XML-based test specification language (§4.1) to describe these scenarios, including various compositions of triggers (§4.2), which permit productive reuse of fault injection scenarios. LFI employs several optimizations in the evaluation of triggers, aimed at reducing runtime overhead (§4.3).

4.1 Description Language

Fault injection scenarios can be written by hand, but we believe practitioners also want to use automated tools for generating and modifying these scenarios (such as the call site analyzer described in §5). For scenarios to be both human-readable and machine-readable, we chose an XML-based language. Here we provide an overview of the language, and direct the interested reader to the complete DTD available at `http://lfi.epfl.ch`.

An injection scenario has two main constructs: trigger declarations and associations between trigger instances and intercepted library functions. A trigger declaration makes a trigger class known to LFI and creates a named trigger instance. An association links a trigger instance to a library function that LFI intercepts, and specifies the conditions under which the triggers should be evaluated.

Consider the earlier example, which aimed to inject faults in `read` only when the corresponding file descriptor is a pipe, the number of bytes requested is between 1 KB and 4 KB, and a mutex is held by the calling thread. The `ReadPipe1K4KwithMutex` defined in §3.1 can be associated with the relevant library calls as follows:

```
<!-- Make the trigger known to LFI -->
<trigger id="readTrig1"
         class="ReadPipe1K4KwithMutex" />

<!-- Invoke the trigger for read() calls -->
<function name="read" argc="3"
          return="-1" errno="EINVAL">
  <reftrigger ref="readTrig1" />
</function>

<!-- Trigger needs to see the lock/unlock calls -->
<function name="pthread_mutex_lock"
          return="unused" errno="unused">
  <reftrigger ref="readTrig1" />
</function>

<function name="pthread_mutex_unlock"
          return="unused" errno="unused">
  <reftrigger ref="readTrig1" />
</function>
```

The `<trigger>` element declares a trigger *instance* identified by the name `readTrig1` and implemented by the class `ReadPipe1K4KwithMutex`. This must either be a C++ class written by the tester, or an LFI stock trigger. The same trigger class can be used for multiple trigger instances.

The `<function>` element creates an association between the `read` library function and the `readTrig1` trigger instance. Whenever `read` is called by the target program, `readTrig1` is asked for a yes/no answer regarding whether to inject a fault or not. To make this decision, `readTrig1` is given by LFI three arguments from the original call stack (`argc` attribute); these arguments correspond to the file descriptor, buffer pointer, and number-of-bytes parameters of the intercepted `read` call. The trigger uses the values of these arguments to determine whether the file descriptor is a pipe and whether the requested number of bytes falls in the target range. If `readTrig1` returns true, then LFI returns to the caller a return value of -1 (`return` attribute) from `read` and sets the `errno` variable to `EINVAL`.

The other two `<function>` associations serve the purpose of informing the trigger about the corresponding mutex lock/unlock calls, giving the trigger instance the opportunity to update its state. Since these associations will never result in injections, the `return` and `errno` attributes are set to "unused."

Triggers can also be parametrized, i.e., the test scenario can specify arguments to be passed to the trigger instance at initialization time. This means, for instance, that one could replace the `ReadPipe1K4KwithMutex` class with a new class that takes the upper and lower bound of the number of bytes as arguments, instead of them being hardcoded to 1 KB and 4 KB. An example of such a class is the `ReadPipe` trigger class in the next subsection.

4.2 Trigger Composition

Triggers can be composed, to form more complex triggers. By default, associating multiple triggers within one `<function>` declaration implies a conjunction of the triggers: they all have to evaluate to true in order for a fault to be injected.

Consider the earlier pipe `read` example: instead of using the `ReadPipe1K4KwithMutex` class, we can use a conjunction of two trigger classes, `ReadPipe` and `WithMutex`. The first one handles injections in the `read` function when the file descriptor is a pipe and the number of bytes requested is between a configurable minimum and maximum. The second one injects a fault in any function, as long as the caller holds a mutex.

Trigger composition allows wider reuse of triggers and, together with parametrization, encourages writing flexible, general triggers. The scenario below illustrates the composition of two triggers, `readTrig2` and `mutexTrig`. The first `<trigger>` declaration illustrates the initialization of the parametrized `ReadPipe` trigger: it allows the tester to specify the upper and lower bound on the number of bytes, and these values are passed to the `Eval` method of the `ReadPipe` trigger.

```
<!-- Declare & initialize a ReadPipe instance -->
<trigger id="readTrig2" class="ReadPipe">
  <args>
    <low>1024</low>
    <high>4096</high>
  </args>
</trigger>

<!-- Declare a WithMutex instance -->
<trigger id="mutexTrig" class="WithMutex" />

<!-- Invoke the composition for read() calls -->
<function name="read" argc="3"
          return="-1" errno="EINVAL">
  <reftrigger ref="readTrig2" />
  <reftrigger ref="mutexTrig" />
</function>

<!-- Trigger needs to see the lock/unlock calls -->
<function name="pthread_mutex_lock"
          return="unused" errno="unused">
  <reftrigger ref="mutexTrig" />
</function>

<function name="pthread_mutex_unlock"
          return="unused" errno="unused">
  <reftrigger ref="mutexTrig" />
</function>
```

Triggers can also be composed into a disjunction, whereby a fault is injected whenever any trigger in the composition returns true; for this, one just adds multiple `<function>` elements using the same function `name`, each one associated with a different trigger. Besides conjunctions and disjunctions, LFI can support negation, whereby the result of a trigger is simply inverted. Using disjunction, conjunction, and negation, one can assemble a wide range of trigger combinations.

4.3 Trigger Evaluation

LFI evaluates triggers in the order in which they appear in the injection scenario. However, in the case of trigger compositions, LFI invokes the smallest number of triggers needed to determine the result of the composition. For example, in the case of conjunctions (i.e., multiple trigger instances referenced in the same `<function>` element), if the first trigger returns false, then the remaining triggers are not invoked at all. This optimization reduces runtime overhead and is similar to the short-circuit evaluation used in C/C++ logical expressions. This feature can be leveraged when composing with the stock singleton trigger: if added to the end of a conjunction, it ensures that a fault is injected once when all the other triggers in the composition return true.

LFI's internal data structures ensure that the list of triggers for the currently intercepted function is obtained in $O(1)$ time, i.e., it is independent of the size of the fault injection scenario. To eliminate runtime overhead during program startup, LFI uses lazy initialization: each trigger is initialized right before it is invoked for the first time.

5 Call Site Analysis

The call site analyzer helps developers find "interesting" places to inject faults, i.e., parts of the target system that are likely to be buggy. The analyzer runs entirely autonomously and looks for call sites where the return values or error side effects of the call are not properly checked. An example is the following code snippet:

```
dir = opendir(pathToDir);
while (ent = readdir(dir)) {
...
}
```

The code works properly most of the time, when `pathToDir` points to an existing directory, but crashes if the directory does not exist or `opendir` cannot allocate sufficient memory. Since the return value of `opendir` is not checked, `readdir` could be passed a null pointer. Although a rather obvious bug, we found similar bugs in widely used software like BIND and Git.

The call site analyzer combs the target program binary for places where a library function is called, and then uses dataflow analysis to determine whether the program checks for all possible errors that the function could return (as indicated by the corresponding library's fault profile, described in §2). The analyzer's method is heuristic, but we found it to be highly accurate, even if not perfect (an occasional false positive is acceptable, given that the potential bug sites can easily be verified through fault injection). While, in theory, disassembling binaries cannot be completely accurate, it has been shown that in commercial-grade applications over 99% disassembly accuracy can be achieved [22].

Once the analysis is complete, the analyzer generates a fault injection scenario targeted at the vulnerable sites with the missing errors; these scenarios employ the generic call stack trigger. The tester would then run a test based on the injection scenarios. The workload for exercising the specific call must be provided by the tester; LFI can help, by analyzing code coverage information from previous executions and indicating whether previously used workloads exercised the target call sites.

Algorithm 1 describes a simplified version of the general workflow for the analysis. The algorithm takes in the executable X, the function of interest F (e.g., `read`), and the set of error codes E, based on the library fault profile. It produces three sets of call sites: the set C_{yes} of sites where the return of F is checked for all error codes, C_{part} where it is checked for only some of the errors in E, and C_{not} where is is not checked for any errors in E.

Lines 1-2 initialize these sets and find the set $callSites_F$ of places in the target binary where F is called.

For each such call site (line 3), we construct a partial control flow graph for the instructions that *follow* the call to F (line 4), in order to determine how the return value and side effects are handled. We empirically found 100 post-call instructions to be sufficient for building the CFG we require. Indirect branches can make the CFG inaccurate, and the current LFI prototype ignores them. This is not a major issue: our analysis of over 9,000 library calls in real-world software revealed that only 0.13% (104 out of 78,292) were indirect branches.

We then perform dataflow analysis (line 5), to follow the propagation of the function's return value through the program. We look at all targets to which the return value is copied and look at all literals to which this return value (or a copy of it) is compared. We iterate through any loops that may occur, as long as the set of copies of the return value increases. In practice, this set typically stabilizes after a few iterations. Our dataflow analysis is intra-procedural; even though other functions may be called to handle errors, the real systems we analyzed always did this after a local check for error conditions.

The result of the dataflow analysis for each call to F consists of sets Chk_{eq} and Chk_{ineq}, corresponding to error codes checked via equality (as in `if (retval==-1)`) and those that are checked via inequality (as in `if (retval<0)`). If all error codes in E are checked by equality, then the call site goes into the set of fully checked calls (lines 6-7). If error codes are checked via inequality, we assume the entire range of error codes is covered (hence the disjunction in line 6). If only some of the error codes in E are checked by equality, then the call site goes into the set of partially checked calls (lines 8-9). If no error codes in E are checked, the site goes into the set of completely unchecked calls, even if error codes outside E are checked (lines 10-11).

Algorithm 1: Call site analysis (simplified)

```
  Input: Target executable $X$, target function
         name $F$, target function error codes $E$
  Output: Set $C_{yes}$ of fully checked calls,
          Set $C_{part}$ of partially checked calls,
          Set $C_{not}$ of completely unchecked calls
1 $C_{yes} := C_{part} := C_{not} := \emptyset$
2 $callSites_F :=$ parse all calls to $F$ in $X$
3 foreach $site \in callSites_F$ do
4     $cfg :=$ construct partial CFG after $site$
5     $<Chk_{eq}, Chk_{ineq}>:=$ dataflow analysis on $cfg$
6     if $Chk_{eq} \supseteq E \vee Chk_{ineq} \neq \emptyset$ then
7         $C_{yes} := C_{yes} \cup \{site\}$
8     else if $Chk_{eq} \neq \emptyset \wedge Chk_{eq} \subset E$ then
9         $C_{part} := C_{part} \cup \{site\}$
10    else
11        $C_{not} := C_{not} \cup \{site\}$
12 return $<C_{yes}, C_{part}, C_{not}>$
```

Due to space constraints, we omit the side-effect analysis done by LFI, which is responsible for verifying whether side effects shown in the fault profile (such as the `errno` variable) are properly checked. The analysis for `errno` checking is virtually identical to the one used for return values. Failing to check particular values of `errno` (e.g., not restarting a system call interrupted with `EINTR`) can compromise the application's robustness.

The call site analyzer produces two sets of fault injection scenarios, one for C_{not} and one for C_{part}. Testers are probably most interested in the former, but after exhausting the bug vulnerabilities related to C_{not}, they can make use of the latter as well. Note that the call site analyzer does not check the correctness of the error handling code, it just relieves humans of some of the burden involved in testing.

6 Implementation

At the heart of LFI is a library call interception mechanism described in more detail in the original LFI paper [18]. LFI creates a shim library that exports stub functions with the same name as the ones being intercepted. On UNIX platforms, we take advantage of the dynamic linker (using the `LD_PRELOAD` mechanism), and on Windows we use Microsoft Detours [14].

A stub function determines the address of the original function and evaluates the triggers provided in the fault injection scenario. If an injection is to be done, the stub gets the return value and side effect to be injected from the injection scenario and injects them. If no injection is to be done, the stub cleans up the stack and jumps to the original function. A stub looks approximately as follows:

```
int LIB_FUNC_NAME(void) {
  static void * (*original_fn_ptr)();
  if (!original_fn_ptr)
    original_fn_ptr = (void* (*)())
                dlsym(RTLD_NEXT, LIB_FUNC_NAME);
  if (eval_triggers(LIB_FUNC_NAME_triggers,
                    lib_function_args)) {
    /* determine return_code, side_effects */
    /* apply side_effects */
    return return_code;
  } else {
    /* return stack and registers to original values */
    __asm__("jmp [original_fn_ptr]");
    /* original function will return to caller */
  }
}
```

Since LFI has no access to source code or documentation to get the prototypes of intercepted functions, the stub functions do not have any arguments. When calling the original function (i.e., no fault injected), the stub merely removes the current frame from the stack (i.e., the one corresponding to the stub) and passes control directly to the original. This has the advantage of not requiring any information about the number of arguments and their type. When having to pass arguments to a trigger, LFI relies on the injection scenario to specify how many arguments are expected on the stack. In our current prototype, we assume all arguments are word-sized.

Designing an extensible trigger system was difficult. Our goal was to allow developers to simply drop the trigger classes in a particular location and then be able to refer to the triggers by their class name in injection scenarios. In other words, we wanted a mechanism similar to Java's `Class.forName()`. We used a variation of the Registry pattern, where each trigger class automatically inherits a factory method and a static member variable whose initialization causes the class name along with the associated factory method to be added to a global map. Instantiating a trigger is done by searching in the map and using the corresponding factory method.

To maximize ease of use, we added to LFI the ability to directly handle DWARF debug information [17]. For example, the call stack trigger allows testers to specify that injections should be done only if execution passes through certain filename/linenumber locations. Another example is the call site analyzer, which can provide the exact source code location of a particularly "suspicious" call, whenever debug symbols are available.

7 Evaluation

LFI's main strength is precision—it allows testers to specify exactly what fault to inject, where to do so, and when to inject. This can be used to selectively inject faults on a particular call when servicing a specific workload, when the program enters a particular state, or when control flow passes through a specific set of points. It is fairly obvious how, combined with knowledge of the system being tested, LFI can be a tester's "power tool."

However, such knowledge may not always available, so we focus our evaluation on how LFI can be used productively even without knowledge of the code. We inject faults in several real systems (§7.1) and find several previously unknown bugs; we also measure the improvement in recovery-code coverage. We measure the accuracy and efficiency of automated injection site identification (§7.2). We then show how LFI can be used to study the behavior of distributed systems and find interesting vulnerabilities (§7.3). Finally, we measure the overhead introduced by LFI triggers and find that their interference with the tested system is negligible (§7.4).

We evaluated LFI on four systems representing four different classes of applications: BIND 9.6.1, MySQL 5.1.44, Git 1.6.5.4 and PBFT 2008-12-09. BIND is perhaps the most popular Domain Name System (DNS) server used in the Internet, being the de facto standard for most UNIX-based network infrastructures. Git is a modern distributed version control system that was initially designed and developed for Linux kernel development, and has experienced tremendous popularity since then. MySQL is a well known and widely used open-source database management system. PBFT [7] is a practical replicated Byzantine fault tolerance system, designed to correctly serve requests in the face of f Byzantine replica failures, as long as there are at least $3f+1$ total replicas. All experiments were run on a 4-core 2 GHz Intel Xeon CPU with 4 GB of RAM, running Ubuntu 9.04 with Linux kernel 2.6.28.

We used the binary distributions of the systems listed above. We resorted to source code only when needed to manually confirm LFI's results.

7.1 Effectiveness of Testing: Bugs and Coverage

When assessing an automated testing tool, there are generally two measures of interest: how many high-impact bugs it finds, and to what extent it improves code coverage. For this section, we run the call site analyzer on the target binaries and directly apply, with no modifications, the injection scenario it generates. Of course, the deeper the knowledge one has of the system being tested, the more effectively LFI's injection triggers can be used. However, here we focus mainly on what can be done entirely automatically, using only stock triggers. We briefly show how human intervention is useful in connecting injected faults to bugs and in writing custom triggers for particular bug categories.

As a first measure of effectiveness, Table 1 lists the 11 previously unknown bugs found by LFI entirely on its own. We expect that, in the hands of a human tester, LFI could find substantially more bugs.

We use the last bug in Table 1 to illustrate the process followed in these experiments. After running on the PBFT binary, the call site analyzer generates an injection scenario, of which we show a fragment below. We then passed this scenario to the LFI injector. Upon inspecting the report of the test, we found that a replica had crashed due to a segmentation fault; the log indicated the id of the trigger that fired in that particular test case. Based on the trigger and an inspection of the source code, we immediately found that the replica's shutdown code attempts to write a checkpoint to a file, without checking that the file has been properly opened.

System	Bug
BIND	***Crash*** if call to `xmlNewTextWriterDoc` fails while a user is retrieving statistics via HTTP [4]
BIND	***Abort*** due to incorrectly handled `malloc` return value in method `dst_lib_init` [3]
MySQL	***Abort*** after a double mutex unlock, due to a failed `close` [19]
MySQL	***Crash*** due to a failed `read` (error code `EIO`) while processing `errmsg.sys` [20]
Git	***Data loss*** caused by running an external command with an incomplete environment, due to failed `setenv` [11]
Git	***Crash*** due to calling `readdir` with a NULL pointer returned by a previously failed `opendir` call [9]
Git	***Crash*** due to unhandled `malloc` return value on line 567 in `xdiff/xmerge.c` [10]
Git	***Crash*** due to unhandled `malloc` return value on line 571 in `xdiff/xmerge.c` [10]
Git	***Crash*** due to unhandled `malloc` return value on line 191 in `xdiff/xpatience.c` [10]
PBFT	***Crash*** caused by a failed `recvfrom` call
PBFT	***Crash*** due to calling `fwrite` with a NULL pointer returned by a previously failed `fopen` call (see below for details)

Table 1: Bugs found automatically by LFI (more details can be found in the referenced bug reports).

```
<trigger id="8054a69" class="CallStackTrigger">
   <args>
   <frame>
     <module>
       bft/bft-simple/simple-server
     </module>
     <offset>
       8054a69
     </offset>
   </frame>
   </args>
</trigger>
<function name="fopen"
   retval="0" errno="EINVAL">
   <reftrigger ref="8054a69" />
</function>
```

The other PBFT bug is interesting, as it does not manifest in the debug build, but only in the release build. Faults were injected in `sendto` and `recvfrom` (simulating deteriorated network conditions), successively in calls made by different replicas (i.e., inject in a call made by replica R_1, then in a call made by replica R_2, etc.). This type of network behavior results in a segmentation fault in the view change phase of PBFT, when the replica tries to access a previously committed message. The reason this bug does not manifest in the debug build is because, when the debug flag is on, the PBFT implementation checks to see if messages were sent correctly and halts with an error code as soon as a problem occurs. The release (i.e., non-debug) build skips this check.

The `malloc` bug in BIND and the `close` bug in MySQL represent interesting cases of buggy recovery code. In BIND, the `dst_lib_init` method checks the return value of `malloc` calls, and runs recovery code if any such call fails. The recovery code destroys the created data structures, by calling `dst_lib_destroy`. The first statement in this method is an assertion, checking that the `dst` data structures have been initialized. However, the call from `dst_lib_init` is made before the `dst_initialized` flag is set, therefore triggering the assertion. In MySQL, the `mi_create` method has error handling code that releases resources, including a particular mutex. However, a failed `close` call can trigger this code after the mutex has already been released by the "normal" program flow, leading to a double unlock and an application crash.

These scenarios illustrate the importance of tools targeted at testing recovery code: such code is hard to exercise in the testing lab without LFI-like tools, and it rarely gets exercised in the field. Yet, whenever it runs, it is expected to run flawlessly.

The second MySQL bug is caused by an uninitialized data structure access after a failed `read`. A related bug, describing a silent failure of MySQL when the `errmsg.sys` configuration file is not found [21], has been acknowledged and fixed. However, if the file exists but reading from it fails for a reason such as a low-level I/O error, MySQL logs the error but nonetheless tries to access an uninitialized data structure and crashes.

When testing MySQL, we started out by using random injection, because MySQL routinely checks function return values, so we wanted to see how robustly it does so. Yet, 1,000 random injection tests targeting different functions caused 35 distinct crashes in MySQL. We analyzed the 35 core dumps and, in this way, found the two bugs presented in Table 1. After writing a specific call stack trigger to reproduce each one of them, we attached a debugger and stepped through the code until the bug manifested; in this way, we were able to connect the injected fault to the bug manifestation.

Custom triggers: To show how triggers can increase testing precision, compared to random injection, we evaluate in Table 2 the precision of three injection scenarios. We report the number of times the `close` MySQL bug presented in Table 1 was activated while running 100 times the `merge-big` MySQL test suite component. This also illustrates step-by-step how to build a custom trigger for a particular category of bugs:

1. The first attempt used random injection, with a 10% injection probability in each `close` call. This approach triggered the bug 16 times. Bigger injection probabilities lower the precision, because faults end up being injected in other `close` calls, and execution does not reach the intended target.

2. In our second attempt, we took advantage of "domain knowledge" and used the call stack trigger to inject faults with a 10% probability only in calls issued from the code in the particular file where the bug resides. This scenario triggered the bug 45 times.

3. For the final scenario, we took advantage of a peculiarity of this bug: the `close` call happens after a mutex unlock. Therefore, we injected faults in `close` calls that happen shortly after a mutex unlock, in the hope that the fault will trigger cleanup code that will cause a double unlock. We created a parametrized trigger that allows specifying the maximum distance, in number of lines of code, between the injection site and the last mutex unlock. This trigger, with a distance of 2, reproduced the bug 100% of the time. This excellent precision illustrates our earlier point that triggers need only be "precise enough."

Trigger scenario	Precision
Random (10%)	16%
Random (10%) within bug's file	45%
Close after mutex unlock	100%

Table 2: Precision of three triggers targeting the `close` MySQL bug from Table 1.

Recovery-code coverage: Improving coverage of recovery code is notoriously hard, because exercising such code typically requires errors that appear outside the scope of the developed program and are hard to simulate. Although scenarios that exercise recovery code are rarely encountered in practice, programs that must operate reliably (e.g., servers) should be able to recover gracefully from such faults without corrupting user data or crashing. Since Git and BIND are mature, widely-used applications, we expect them to have recovery code for a large set of possible environment errors.

To assess the coverage improvements that LFI can achieve, we first used gcov and lcov to measure the level of recovery-code coverage obtained by the test suite that ships with each of the applications. We manually identified in the lcov results the recovery code blocks for the functions we target for injection—a tedious job, but necessary for an accurate comparison. We then ran the LFI call analyzer on the two target applications; to be conservative, we trimmed the resulting injection scenarios down to approximately 25 library function calls that are known to fail on occasion (e.g., `fopen`, `read`, `sendto`, `fstat`) and excluded all others. We re-ran the default test suite and measured the new level of coverage. Table 3 shows the results.

	Git	BIND
Additional recovery code covered	~35%	~60%
Additional LOC covered by LFI	429	560
Total coverage without LFI	78.7%	61.2%
Total coverage with LFI	79.6%	61.8%

Table 3: Automated improvement in code coverage.

The fact that, without any human assistance, LFI was able to make the default test suite cover up to an additional 60% of the recovery code in mature applications suggests LFI can offer substantial benefits to testers out-of-the-box. The numbers reported in Table 3 are only a conservative estimate of the improvement in testing, because (a) we did not write any new tests, rather relied on the workload generated by the default test suite; (b) we did not test any of the calls for which there was no recovery code at all, even if there should have been; and (c) we injected faults in only a subset of the library calls made by the applications. Note that the call site analyzer can suggest targets for additional tests, thus helping test developers write tests with less effort.

7.2 Injection Target Identification: Accuracy and Efficiency

There are two ways to identify good injection targets: manually or automatically. We believe the most practical approach is one in which injection targets are first identified automatically, by tools like the LFI call site analyzer, and then developers manually refine the generated injection scenarios. The refinement can be done either based on knowledge of the target system or iteratively, by trying out increasingly focused failure scenarios of interest.

To maximize usefulness of an automated injection target identifier, it must be accurate. The accuracy of injection target identification can be defined as:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

where TP stands for the number of true positives, TN for true negatives, FP for false positives, and FN for false negatives. In the context of injection target identification, these are defined by the following confusion matrix:

LFI says...	Actually checked	Not actually checked
error return is checked	TN	FN
error return is not checked	FP	TP

We used the call site analyzer to identify places in the target system where libc calls are made and the return code is not checked. We then manually inspected the code to cross-check the results (see Table 4). Note that we did not specifically select the ones that are favorable to LFI; we are showing here *all* the calls for which we performed the manual inspection and validation.

System	Function	TP+TN	FN	FP	Accuracy
BIND	malloc	17	0	0	100%
BIND	unlink	6	0	0	100%
BIND	open	5	0	1	83%
BIND	close	39	0	0	100%
Git	malloc	25	0	0	100%
Git	close	127	0	0	100%
Git	readlink	7	0	0	100%
PBFT	fopen	6	0	0	100%

Table 4: LFI's call site analysis accuracy with no human assistance, no documentation, and no source code.

Based on these results, we conclude that LFI's call site analysis is highly accurate for libc calls, even though it is performed directly on x86 binaries; we expect this accuracy to carry over to other libraries beyond libc. It is therefore reasonable to expect that LFI can automatically provide a good set of injection scenarios that developers can then adjust as needed for their tests.

Efficiency: Besides accuracy, running time of the analyzer is also an important factor, because testers are unwilling to wait long for results. For example, it is frequently said that the long running times of model checkers have discouraged their widespread use in testing.

The LFI call site analyzer is fast: in our experiments, analysis time ranged from 1 second to a maximum of 10 seconds for BIND, in cases where there were more than 100 call sites. Analysis time is only influenced by program size (i.e., number of machine instructions) and number of call sites that have to be analyzed.

Developers can process the results of the analyzer fairly quickly. With each call site found, the details regarding file name and line number are provided, if debug symbols are available; this information can guide the developer in inspecting the source code.

7.3 Studying System Behavior

Finding bugs is not the only objective of a tool like LFI—it can also be used to study the behavior of systems under various circumstances. For example, the users of a distributed system may be interested in knowing how it behaves in the face of network failures. We illustrate here the use of LFI for studying the behavior of PBFT's implementation, which is hard to reason about based solely on the design, without experimental evaluation. Our setup consisted of four replicas (i.e., $f = 1$) and one client. We used the *simple_client* and *simple_server* programs shipped with PBFT to generate test workload.

In our first experiment, we used LFI along with a stock distributed trigger (§3.2) to see how PBFT's performance is affected by faults in inter-replica communication. We randomly injected faults in `sendto` and `recvfrom` with a variable probability, simulating a degraded (but not malicious) network environment. Using as a baseline PBFT's performance without LFI's interference, we show in Figure 3 how the slowdown varies (averaged over 7 trials) as network conditions worsen. The performance of PBFT deteriorates gradually, reaching a maximum of 4.17× slowdown for a 99% probability of packet loss (i.e., when only one in every 100 network messages make it to the receiver).

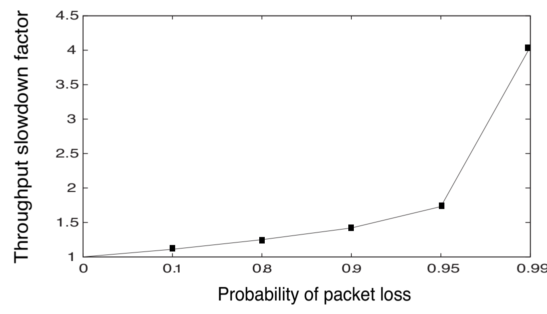


Figure 3: Slowdown in PBFT under progressively worsening network conditions.

While the trend of the curve is not surprising, the exact amount of slowdown experienced at every level of degradation would be difficult to guess without direct measurement.

In a second experiment, we used LFI to simulate a denial-of-service (DoS) attack on PBFT's replicas and we measured again PBFT end-to-end performance, averaged over 7 trials. The baseline for the experiment corresponds to LFI intercepting the calls but letting them all succeed. By injecting consecutive faults in all communication of a specific replica (thus rendering it practically inactive), we obtained an overall performance *improvement* of 12%, possibly due to reduced communication overhead. The third set of measurements corresponds to an attack in which 500 consecutive faults were injected in replica R_1's communication, then 500 in R_2's, then 500 in R_3's, then again 500 in R_1's, and so on. Such an attack targets the reconfiguration protocol, aiming to confuse it. PBFT's throughput dropped by a factor of $2.2\times$.

The results indicate that the second DoS attack scenario is more effective than the first. While this behavior may not be necessarily surprising, the number of faults and their impact on the overall performance cannot be easily inferred from the design of the system.

As seen in this section, LFI can be used to study the behavior of system implementations under various failure conditions. Since LFI works on binaries, we believe this can be a useful tool for engineers who wish to evaluate, for instance, closed-source third party software, such as databases, load balancers, or application servers.

7.4 The Precision/Performance Tradeoff

It is obvious that LFI triggers can be designed at arbitrary levels of precision, using information in call stacks, program variables, system state, etc. In the context of fault injection, precision denotes the degree to which repeated runs of the target program trigger the same injections. For example, a precise injection would be one that is made only when the system processes a specific request (e.g., a database answering a SQL query), but not when it processes other queries, even if the same library functions are called in the same conditions.

The question we wish to answer is: What is the cost of this precision? If, for instance, the process of injecting library-level faults via LFI slows down the system to the point that its behavior is no longer representative, then the value of testing is decreased (though not eliminated).

To analyze the performance of the trigger mechanism, we measured two commercial-grade servers that are highly performance-sensitive: the Apache 2.2.14 Web server and the MySQL 5.1.44 database server. We computed the induced overhead as a function of number of triggers, frequency of triggering, and type of triggers. We used the Apache benchmark (AB) [1] on Apache and the SysBench [25] Online Transaction Processing benchmark on MySQL.

In order to allow the benchmarks to proceed correctly, we did not actually inject faults, but allowed the triggers to pass the calls through to the real library functions. In this way, we focus the measurement on the triggering mechanism. Our purpose is not to measure how long it takes the applications to recover after encountering a fault, but rather what overhead is introduced by LFI's trigger mechanism.

We constructed injection scenarios with a variable number of triggers and combinations thereof:

- Trigger 1: This trigger targeted `apr_file_read` calls. We used it for targeting calls that have the file descriptor pointing to a socket. The trigger uses the `apr_stat` function on the received file descriptor to check its type.

- Trigger 2: As we wanted to focus testing on Apache's core, and exclude dynamically loaded modules, we also decided to check if the function caller is Apache via the call stack trigger.

- Trigger 3: We further narrowed down our injection target by requiring that the function call happens while processing a request. We used a variation of the stock call stack trigger to require the existence of Apache's `ap_process_request_internal` function in the call stack.

- Trigger 4: We then adapted the stock application state trigger in order to permit injections only when the HTTP *POST* method is used to make the request. To do so, the trigger had to analyze the `request_rec` argument received by Apache's `ap_process_request_internal` function and examine its `method_number` field.

- Trigger 5: We wrote a custom trigger in order to intercept lock and unlock methods in order to target only `apr_file_read` calls made while holding a mutex.

Table 5 summarizes the results, obtained on two different benchmark workloads: static HTML and PHP requests. The former, consisting of 1,000 requests of a static HTML page, fires triggers 32,612 times (about 1.7×10^5 triggerings/second) for the maximum number of five triggers. The second workload is computationally more demanding on Apache, resulting in fewer library calls per unit of time: a total of 45,228 triggerings (about 2.8×10^4 triggerings/second). In both cases, the overheads introduced by trigger evaluation are negligible, suggesting that LFI can be used without affecting the target system's behavior other than through the injected faults.

	Static HTML	PHP
Baseline (no LFI)	0.179 sec	1.562 sec
1 trigger	0.179 sec	1.564 sec
2 triggers	0.179 sec	1.574 sec
3 triggers	0.179 sec	1.577 sec
4 triggers	0.186 sec	1.585 sec
5 triggers	0.188 sec	1.589 sec

Table 5: Running time of the Apache Web server while using LFI with 1-5 triggers. The baseline represents Apache httpd without any interference from LFI.

We also ran the SysBench [25] Online Transaction Processing (OLTP) benchmark on the MySQL RDBMS with LFI applied to GNU libc. We devised injection scenarios with 1-4 triggers on the `fcntl` function:

- Trigger 1: Inject when the `cmd` argument is `F_GETLK`.

- Trigger 2: Inject only when the thread count is bigger than 64 (tests the global variable `thread_count` with the application state trigger).

- Trigger 3: Inject only when the system is shutting down (tests the global variable `shutdown_in_progress` with the application state trigger).

- Trigger 4: Inject when the call is made by the main application module and not other libraries (uses the call stack trigger).

Table 6 shows the results for random triggering and two different workloads: read-only and read-write queries. For the highest number of four triggers, there were ~14K triggerings/second.

	Read-only	Read/Write
Baseline (no LFI)	1076 txns/sec	326 txns/sec
1 triggers	1064 txns/sec	319 txns/sec
2 triggers	1060 txns/sec	318 txns/sec
3 triggers	1056 txns/sec	316 txns/sec
4 triggers	1056 txns/sec	316 txns/sec

Table 6: MySQL database server performance while applying LFI with 1-4 triggers (number of transactions per second, as reported by the SysBench OLTP benchmark).

Even with complex combinations of conditions that check various parts of the system state, LFI introduces negligible overhead (consistently less than 5% in our measurements). This offers an advantageous precision/performance tradeoff, meaning that testers can afford to use sophisticated triggers without being concerned that trigger evaluation will bias system behavior.

8 Related Work

Library-level fault injection is an inexpensive testing method first proposed in the context of FIG [6], a tool used to verify the error handling code responsible for GNU libc errors. FIG has several important limitations, in that it only allows injecting faults in GNU libc functions, cannot select particular call sites in which to inject, and requires hardcoding the injected error values.

A refinement was presented by Süßkraut & Fetzer [24] in the form of a system that finds bugs via library-level fault injection and then patches the vulnerable applications to protect against these bugs. Still, this system is limited to GNU libc functions and does not have the means to automatically search for vulnerable call sites, nor to specify complex injection conditions.

This paper continues our previous work on LFI [18], a tool that enables a more general approach to library-level fault injection, by automatically determining meaningful faults to inject and by supporting the interception of arbitrary library functions without the need for source code. To our knowledge, this was the first library-level fault injector practical enough for real-world use. Its main disadvantage, however, proved to be the lack of a mechanism for specifying precise injection conditions, leaving the tester able to only do random or exhaustive exploration of the fault space. The work presented here addresses this shortcoming.

Ideas similar to call site analysis have been previously proposed in [12], where the authors targeted Linux file system implementations at the source-code level and used block-level fault injection to confirm certain categories of bugs. Java exception propagation and handling has been analyzed by Weimer and Necula [27] and Fu et al. [8], using functional specifications and compile-time fault injection, respectively, for discovering bugs. Our approach is complementary, since we target different types of systems, use fault injection at a different level, and operate on binaries.

Other fault injection systems, like G-SWFIT [2], follow another approach to testing: they mutate the target binary code according to statistical bug rules. Different tools operate at even lower levels: FTAPE [26] is designed to inject faults in memory, registers, and disk accesses. NFTAPE [23] can use different low-level fault injectors to test the robustness of systems. Using these systems for testing general-purpose software faces two challenges: it is not clear whether it is reasonable to expect an application to handle such low-level fault (e.g., disk failure), and the large number of layers that separate the low-level injection point from the application level makes pinpointing the location of a possible bug tedious.

The application-library boundary does not suffer from these shortcomings: programs are expected to react properly to error conditions signaled by components with which they interact, and determining the point where a fault transformed into an error is easier, albeit not trivial.

The concept of injection triggers was used by FERRARI [16] and other tools for OS robustness evaluation [15]. These early trigger mechanisms could only specify a predefined set of conditions, like injecting after the *n*-th function call or after a determined amount of time. The LFI stock triggers and the `Trigger` interface, however, allow testers to achieve a level of precision in recovery-code testing that was previously not available.

9 Conclusion

This paper described a new and improved version of LFI, a library-level fault injection framework that is able to automatically identify errors externalized by shared libraries, identify potentially vulnerable injection targets in application binaries, and produce injection scenarios that exercise such vulnerabilities. The new LFI offers an injection triggering mechanism that allows testers to specify with high precision the conditions under which a fault is to be injected. We presented the stock triggers provided by LFI and the mechanism through which they can be extended to fit practitioners' needs.

LFI was successfully used in testing real systems, and it found 11 new bugs in the BIND name server, the Git version control system, the MySQL database server, and the PBFT replication system. LFI achieved 35%-60% recovery-code coverage entirely on its own, with no human involvement. We have also shown that LFI introduces only negligible runtime overhead during testing.

LFI can be downloaded at `http://lfi.epfl.ch/`.

Acknowledgments

We wish to thank our shepherd, Andrew Baumann, the USENIX anonymous reviewers, and our EPFL colleagues for their valuable help in improving our paper.

References

[1] Apache Benchmark (AB). `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] R. Barbosa, N. Silva, J. Duraes, and H. Madeira. Verification and validation of (real time) COTS products using fault injection techniques. *Intl. Conf. on Commercial-off-the-Shelf (COTS)-Based Software Systems*, 2007.

[3] BIND - [BUG] BIND abort in dst_api.c. `https://lists.isc.org/pipermail/bind-users/2010-January/078493.html`.

[4] BIND - [BUG] BIND crash in statschannel.c. `https://lists.isc.org/pipermail/bind-users/2010-January/078428.html`.

[5] E. Bisolfati, P. D. Marinescu, and G. Candea. Studying application–library interaction and behavior with LibTrac. In *Intl. Conf. on Dependable Systems and Networks*, 2010.

[6] P. A. Broadwell, N. Sastry, and J. Traupman. FIG: A prototype tool for online verification of recovery mechanisms. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems*, 2002.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symp. on Operating Systems Design and Implementation*, 1999.

[8] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, 2004.

[9] Git - [BUG] crash on make test. `http://marc.info/?l=git&m=125985479417107`.

[10] Git - Git unchecked mallocs. `http://marc.info/?l=git&m=126298802319662`.

[11] Git - Running commands in wrong environment. `http://marc.info/?l=git&m=125986795807036`.

[12] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit. EIO: error handling is occasionally correct. In *USENIX Conf. on File and Storage Technologies*, 2008.

[13] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Symp. on Operating Systems Design and Implementation*, 2008.

[14] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *USENIX Windows NT Symp.*, 1999.

[15] A. Johansson, N. Suri, and B. Murphy. On the impact of injection triggers for OS robustness evaluation. In *Intl. Symp. on Software Reliability Engineering*, 2007.

[16] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2), 1995.

[17] libdwarf. `http://reality.sgiweb.org/davea/dwarf.html`.

[18] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *Intl. Conf. on Dependable Systems and Networks*, 2009.

[19] MySQL - [BUG] MySQL crash due to double unlock. `http://bugs.mysql.com/bug.php?id=53268`.

[20] MySQL - [BUG] MySQL crash due to error while reading errmsg.sys. `http://bugs.mysql.com/bug.php?id=53393`.

[21] MySQL - [BUG] MySQL crash due to missing errmsg.sys. `http://bugs.mysql.com/bug.php?id=25097`.

[22] M. Prasad and T. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conf.*, 2003.

[23] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Intl. Computer Performance and Dependability Symp.*, 2000.

[24] M. Süßkraut and C. Fetzer. Automatically finding and patching bad error handling. In *European Dependable Computing Conference*, 2006.

[25] Sysbench. `http://sysbench.sourceforge.net`.

[26] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Intl. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, 1995.

[27] W. Weimer and G. C. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems*, 30(2), 2008.

Sleepless in Seattle No Longer

Joshua Reich[†], Michel Goraczko, Aman Kansal, Jitendra Padhye

[†]*Columbia University, Microsoft Research*

Abstract: In enterprise networks, idle desktop machines rarely sleep, because users (and IT departments) want them to be always accessible. While a number of solutions have been proposed, few have been evaluated via real deployments. We have built and deployed a lightweight sleep proxy system at Microsoft Research. Our system has been operational for six months, and has over 50 active users. This paper focuses on providing a detailed description of our implementation and test deployment, the first we are aware of on an operational network.

Overall, we find that our lightweight approach effected significant energy savings by allowing user machines to sleep (most sleeping over 50% of the time) while maintaining their network accessibility to user satisfaction. However, much potential sleep was lost due to interference from IT management tasks. We identify fixing this issue as the main path to improving energy savings, and provide suggestions for doing so. We also address a number of issues overlooked by prior work, including complications caused by IPsec. Finally, we find that if certain cloud-based applications become more widely adopted in the enterprise, more specialized proxy reaction policies will need be adopted. We believe our experience and insights will prove useful in guiding the design and deployment of future sleep solutions for enterprise networks.

1 Introduction

A number of studies [30, 3, 41, 6] have noted that most office machines are left on irrespective of user activity. At Microsoft Research, we find hundreds of desktop machines awake, day or night – a significant waste of both energy and money. Indeed, potential savings can amount to millions of dollars per year for large enterprises [40].

As businesses become more energy conscious, more desktops may be replaced by laptops. However, currently desktops comprise the majority of enterprise machines [18], with hundreds of millions additional desktops being sold every year [22, 27, 18]. Where users make heavy use of local resources (e.g., programming, engineering, finance), desktops continue to be the platform of choice. Hence, managing desktop power consumption is an area of both active research [30, 3, 41, 6] and commercial [2, 39, 33] interest.

The most common reason that desktops are kept idling is that users and IT administrators want remote access to machines at will. Users typically want to log into their machines or access files remotely [3], while IT administrators need remote access to backup, patch, and otherwise maintain machines. A number of solutions to this problem have been proposed [33, 3, 6, 30]. The core idea behind these is to allow a machine to sleep, while a *sleep proxy* maintains that machine's network presence, waking the machine when necessary. Some of these proposals rely on specialized NIC hardware [33, 3]; others advocate use of network-based proxies [6, 30].

Unfortunately, most previous work has been evaluated either using small testbeds [3, 30, 6] or trace-based simulations [30]. We are not aware of any paper detailing the deployment of any of these proxying solutions in an operational enterprise network on actual user machines.[1] This is disconcerting: systems that work well on testbeds often encounter potentially serious challenges when deployed in operational networks.

This paper aims to fill that gap. We describe the design and deployment of a network-based sleep proxy on our corporate network. Our design expands on the light-weight network proxy approach proposed in [6, 30], avoiding hardware modification [3] and the overhead of virtualization [17]. Our architecture comprises two core components: a per-subnet sleep proxy, and a sleep notifier program that runs on each client. The sleep notifier alerts the sleep proxy just before the client goes to sleep. In response, the proxy sends out ARP probes [13] to ensure that all future traffic meant for the sleeping client is delivered to the proxy instead. The proxy then monitors this traffic applying a *reaction policy*: responding to some packets on the client's behalf, waking the sleeping client for certain specified traffic (using Wake-on-LAN (WOL) [43] packets), and ignoring the rest. Our reaction policy of waking for incoming TCP connection attempts on listening ports was chosen both in keeping with our goal for a light-weight, easily deployable system and based on the performance predictions of [30]. We provide in-depth discussion of the merits of this and alternative approaches in Sec. 3.

Our system has been operational for over six months, and currently has over 50 active users. Our software is deployed on user's primary workstations, not test machines. Indeed, this preliminary deployment has been so successful that our IT department has begun recommending our system to users. We have instrumented our system extensively; capturing numerous details about sleep and wakeup periods, why machines wake up and *why they stay up*. Instead of using generic estimates of PC power con-

[1]Concurrent work [4], provides the first study of sleep proxy deployment in an academic network.

sumption, we use a sophisticated *software*-based, model-driven system, *Joulemeter*, to estimate power draw.

In this paper, we focus on providing a **thorough description of our implementation and its performance in the wild**. We describe a number of practical issues we encountered when deploying a light-weight sleep proxy in a corporate network, many of which have been overlooked by previous work. For example, our implementation must not only deal with vanilla IPv4 and IPv6 packets, but also tunneled packets. Our corporate network uses IPsec, and we find that *a seemingly minor implementation choice* in this setup, almost entirely *ameliorates the overhead* of dealing with this traffic. We describe race conditions that arise when the sleep proxy attempts to redirect traffic from sleeping client to itself, and provide a practical solution. We show how issues such as DHCP lease expiration and proxy failure can be handled *without the need for a more complex reaction policy*.

The highlights of our deployment experience and performance assessment are:

A light-weight system using a simple reaction policy can produce significant savings. By analyzing trace data from our system, we find that our system allowed the clients to sleep quite well. Many machines slept over 50% of the time, despite use of a simple reaction policy. However, the average power savings was only 20%, casting a pall over the optimistic predictions made in [30, 3].

IT servers and applications proved a major impediment to sleep. The main cause of reduced power savings in our enterprise network was due to the IT setup. We find that while users do access their machines remotely, remote accesses by IT applications are the primary cause of both machines being woken (*fitful sleep*) and being kept awake (*insomnia*). IT server connection attempts repeatedly woke sleeping machines. In one extreme case, a single machine was contacted over 400 times within a two-week period. Additionally, some of the locally running IT applications (e.g., virus scanners) kept machines up by temporarily disabling sleep functionality. We also identify bugs in common software (e.g., Adobe Flash player) that interfere with proper sleeping. Fortunately, it appears there is significant room for improving the compatibility of IT setup and effective sleep. We discuss IT setup impediments and remedies further in Sec. 7.6.

The rise of cloud-based applications may demand more complex reaction policies. Three of our users required support for two popular cloud-based applications, LiveMesh and LiveSynch. Machines running these, or similar, applications must initiate TCP connections to the cloud server, which are used to inform them of any pending updates. These connections can either be periodic, or long-lived, but it must be initiated by client. We refer to cloud applications of this type as *persistent*[2]. Consequently, to support such cloud-based applications the sleep proxy will need to keep some additional state which may be as simple as sending TCP keep-alives or as complex as running a virtualized client-side of the application. While this did not pose a major issue in our operational environment/population, as/where the predominance of persistent cloud applications increases, reaction policies supportive of this model will be needed.

Overall, we believe that the insights gleaned from our experience will be useful in guiding the design and deployment of future sleep solutions in enterprise networks.

[2]Many cloud-based applications including most "software as a service" applications (e.g., Google Docs) are not of this type

2 Related work

While the basic concept of sleep proxying has been known for some time [15], it has received much renewed attention lately [6, 29, 3]. Among recent publications, the two most closely related to our work are [3] and [29].

In [3], the authors describe a hardware-based solution. They augment the NIC with a GumStix [20] device, which is essentially a small form factor, low-powered PC. Once the host machine goes to sleep, the GumStix device takes over. It handles select applications (e.g., file downloads) on behalf of the host PC, but wakes up the host PC for more complex operations. While this approach is more flexible than the sleep proxy we have built, it is far less practical for two reasons. Not only is additional hardware required on every PC, but both applications and host OS modification are required to enable state transfer between host PC and GumStix device. Both these requirements are a substantial barrier to widespread deployment of this technology. In contrast, our approach requires neither extra hardware, nor application modifications.

In [29], the authors carry out an extensive trace-based study of network traffic, arguing for a network-based sleep proxy. Their primary finding is that in an enterprise environment, broadcast and multicast traffic related to routing and service discovery cause substantial network 'chatter", most of which can be safely ignored by a sleep proxy. They also posit that most unicast traffic directed to a host after it has gone to sleep can also be ignored, so long as the host is woken when traffic meant for a set of pre-defined applications arrive (early work had focused on avoiding disrupting existing TCP connections [14, 23]). Based on these insights, they propose a number of sleep proxy designs.

While our proxy design builds upon the insights of [29], we make several additional contributions in this paper. First, unlike [29], our design includes a client-side agent, which considerably simplifies the overall architecture, making it robust, and virtually configuration-free. Second, we build and deploy our sleep proxies in a real operational network on users' primary workstations. In contrast, the prototype in [29] was tested only a small testbed without real users, and did not address challenges such as IPsec traffic and proxy failures. Third, our instrumentation measures sleep and wakeup behavior of operational machines. We document why machines do not sleep, when and why they wake, etc. Fourth, our deployment includes a model-based power measurement competent. Since machine power usage can vary by 2.5x while awake, our power estimates provide significantly greater fidelity than the "one size fits all" model used by [29].

Two pieces of concurrently published work address alternative sleep proxying architectures that make use of a networked sleep proxy. [4] implements a stub-based reaction policy along the lines of [3] and evaluates it in an academic network, while [17] runs client machines within a hypervisor and migrates these to the sleep proxy machine. We provide further comparison in Sec. 3.3.

We now turn to commercial systems. Intel offers two hardware-based solutions, Remote Wakeup Technology (RWT) [33] and Active Management Technology (AMT) [7], that can remotely wake up a sleeping machine. AMT is primarily meant for management tasks (e.g., out of band access for asset discovery, remote troubleshooting). RWT is more closely related to our work. RWT requires the NIC of the sleeping machine to maintain a persistent TCP connection to an authorized server. The NIC wakes up the host machine upon receiving a special message over this TCP connection. RWT requires modification of client applications and works only with Intel hardware. Even the wakeup service has to be digitally signed by Intel. In contrast, our solution is entirely software-based, hardware-agnostic, and requires no application modification.

Apple has recently released a sleep proxy geared toward home networks that works only with select Apple hardware [8]. For enterprise networks, systems such as Adaptiva [2] and Verdiem [39] are available. The primary focus of these systems is to enable the system administrator to estimate power usage, and wake up sleeping machines to perform management tasks such as patching. A number of industry participants are trying to standardize basic sleep proxy functionality [16].

Several other approaches to saving power, such as power-proportional computing [9], dynamic voltage and frequency scaling [34], the TickLess kernel [37] and OS-level power management [35] have been investigated, and can be used in conjunction with our system. Researchers have also looked at networking hardware and software stacks as potential targets for power savings. Examples include [21, 30, 12, 11]. [5, 38, 26] examine data center power consumption and savings approaches.

Prior work has shown that CPU utilization and certain performance counters can be used to estimate computer energy use [32, 10, 19]. Our power estimation technology provides enhanced accuracy by considering additional factors not considered in prior work, such as processor DVFS states and monitor power.

3 Design Goals & Alternatives

As discussed earlier, enterprise users often do not let their machines sleep as they may require remote access. Our goal in deploying a sleep proxy is to encourage users to allow their machines to sleep – by ensuring their machine will wake on remote access attempts. We now describe the basic functionality required from a sleep proxy, define our design goals, and describe design alternatives. Before we begin, we note that our use of the term "sleep" refers to ACPI S3 (suspend to RAM) [1]. Our system supports ACPI S4 (hibernate) and S5 (power off) as well.

3.1 Basic sleep proxy functionality

A *sleep proxy* detects when a *sleep client* machine (M) has gone to sleep, typically because that machine's *idle timeout* had been reached.[3] The proxy then monitors network traffic destined for M. Based on a pre-defined *reaction* policy, the sleep proxy will, (a) respond to some of the traffic on behalf of M (e.g. ARP requests for M), (b) wake M for selected traffic (e.g. TCP SYNs for M) and (c) ignore the remainder.

3.2 Design goals

Our goal is to build a practical, deployable sleep proxy for typical corporate networks, composed of desktop machines with wired connectivity. In a typical usage scenario, the user's machine goes to sleep, and wakes automatically on remote connection attempts.

The design of our sleep proxy was directed by four goals. (a) The system had to save as much power as possible, (b) while minimizing disruptions to users. It is critically important to ensure a sleeping machine is always woken when the user desires access: otherwise no one would use the system. Furthermore, the system had to be (c) easy to deploy and maintain, since we operated without the benefit of a large IT staff. We explicitly decided not to add hardware to client machines, as it makes deployment significantly harder. Finally, we required the architecture be (d) scalable and extensible, since the system had to operate in a dynamic live network

It was not our goal to support laptops *per se* as they offered much less opportunity for power savings. They consume much less power when active, and are more often put to sleep by users. Thus, while some of the work we have done is applicable to laptops, we do not address laptop-specific challenges such as mobility in our work.

[3]In Windows, the idle timeout is typically 30 minutes from power up / last user activity, and two minutes for any other wake cause (e.g., scheduled wakeup, WOL).

As all the machines in our network run Windows, some details of our implementation are Windows specific. However, our architecture is designed to be OS agnostic.

3.3 Design alternatives

We now consider three design alternatives, and evaluate them in light of our requirements.

3.3.1 NIC Pattern Matching

The first potential approach is to simply use the combination of *Wake-On-Pattern+ARP Offload*. This capability is available on most modern wired NICs.
How it works: The NIC effectively acts as the sleep proxy for the machine. It responds to incoming ARPs on behalf of the sleeping machine (*ARP Offload*), thereby maintaining the machine's network presence. The NIC can be programmed to detect specific patterns in incoming traffic, and wake up the host machine if a packet with specified pattern arrives (*Wake-On-Pattern*). The interface for specifying patterns [28] includes built-in support for IPv4 and IPv6 TCP-SYNs; one only need specify additional information (e.g., ports). Raw bit patterns can also be specified.
Pros: These NICs are available on most modern machines, so no additional hardware needs to be deployed.
Cons: We found that for our purposes the capabilities offered by these NICs were not adequate. Our corporate network is quite complex: it supports IPv4, IPv6, v6-over-v4 and requires IPsec. To ensure machines were woken whenever users required access, we had to handle packets requiring flexible inspection (e.g. a TCP SYN in an ESP packet carried in an IPv6 packet, tunneled in an IPv4 packet - Sec. 4.3.2). While such packets may be detected by explicit bit-pattern matching, the number of wakeup patterns needed is a multiple of the number of listening ports (to detect tunneled variations) plus several base patterns for standard WOL functionality. NICs on older machines can support as few as four wake patterns and are limited to detecting matches in the beginning of the packet which restricts the ability to detect tunneled packets. Moreover, future needs (e.g., support for persistent cloud applications) may dictate stateful reaction policies or deeper packet inspection, beyond current NIC capabilities. Thus, this approach fails criteria (b) and (d).

3.3.2 Virtualization

How it works: Users install a hypervisor on their desktop, and then install and use a VM on top of the hypervisor[17]. When the desktop machine needs to sleep, the VM is migrated to a hosting server. When necessary, (e.g. a CPU intensive application is run), the desktop machine is woken, and the VM migrated back.
Pros: This approach is attractive because if the migration can be made seamless, the desktop does not have be woken up for transactions of even moderate complexity that can be carried out on the hosting server. As the machine can go to sleep without interrupting existing network connections, the machines can go to sleep much more often, and hence the power savings may be greater.
Cons: To deploy a system based on this approach, we would have had to install hypervisor on the end user systems and boot their existing OS as a VM. Most users would not have agreed to such a drastic change to their work environment. Apart from taking a performance hit, virtualization may encounter problems with a number of common end-user devices (e.g., cameras, external drives), whose drivers do not always work well when virtualized.

3.3.3 Network-Based Sleep Proxies

This approach was proposed in [15], its feasibility recently given careful study by [29].
How it works: This approach relies on a separate machine acting as a sleep proxy for the sleeping machine. The sleep proxy detects when a client goes to sleep. It then modifies Ethernet routing (Sec. 4.3.1) to ensure that all packets destined for the sleeping machine are delivered to the sleep proxy instead. The proxy examines the packets, and wakes up the sleeping client when needed, by sending a Wake-On-LAN (WOL) [43] packet.
Pros: Very little hardware support is required from the client machine - the client NIC only needs to support WOL. As the sleep proxy runs on a separate, general purpose computer, it has great flexibility in handling incoming traffic for the sleeping machine. The sleep proxy can do complex, conditional packet parsing and can even wake the sleeping machine based on non-network events such as requests by system administrators, users entering the building (with support from building access systems), etc. This design also scales well (Sec. 7.5.2).
Cons: This design requires deployment of a sleep proxy on a separate machine (generally one per subnet supported). In most variations a client-side application must be installed as well.

We have chosen this approach as it is both very easy to deploy and requires minimal changes to user machines. It affords great scalability and flexibility as the sleep proxy can be changed without disturbing the client machines. We have chosen to use light-weight reaction policy which simplifies both client and proxy software complexity and allows a very large number of hosts to be handled by a single proxy. This reaction policy does cause existing network connections are broken. We argue that this is not an issue for typical corporate workloads (Sec. 4.4), although this may change if persistent cloud computing applications play a greater role in corporate environments.

Contrastingly, [4] uses a stub-based reaction policy, capable of maintaining existing network connections and waking the host somewhat more infrequently. This comes at the cost of implementation complexity and will allow fewer clients to be hosted on a sleep proxy. Their implementation uses an ESX server that would preclude either low-power sleep proxies (Sec. 6) or peered proxying (Sec. 8). Their reaction policy faces the same impediments from sleep-unfriendly IT setups as ours - by far the main source of lost sleep opportunities in our environment - as IT tasks generally require waking clients.

4 Architecture

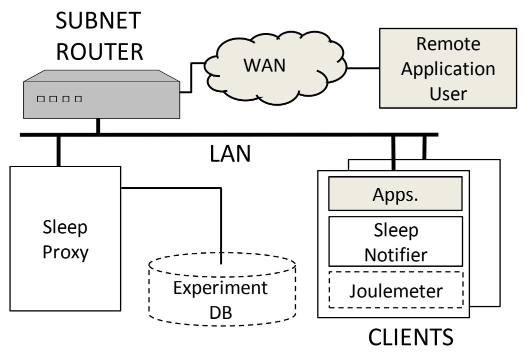


Figure 1: System block diagram. Blocks shaded gray represent existing components that are not modified in any way for the sleep proxy to work. Blocks with dashed outlines are part of our instrumentation setup.

The overall architecture of our system is shown in Figure 1. We require one sleep proxy per subnet. We also required the clients to install a small background service[4], *sleep notifier*. In this section, we will focus on the design of the sleep proxy and the sleep notifier which form the core of our solution. We discuss Joulemeter in Sec. 5.1

As discussed earlier, a sleep proxy responds to some traffic, wakes the sleep client for other traffic, and ignores the rest. Our choice of reaction policy is similar to that of the proxy scheme (*proxy3*), which [29] found provided the highest simulated power savings. This reaction policy, whose rationale is discussed in Sec. 4.4, responds mechanically to IP resolution requests (e.g., ARP) and wakes the sleep client only on TCP connection attempts to listening ports[5], ignoring other traffic.

Before digging into design details (Secs. 4.2 and 4.3), we provide a quick overview of how our system works.

4.1 System Overview

Imagine a *sleep client* M running sleep notifier. M's sleep notifier registers with the OS to receive notification when the machine is about to go to sleep. At such time, the OS alerts the sleep notifier.

[4]A daemon, in Unix terminology.

[5]There being no reason to wake the machine for connections to non-listening ports, which would just be ignored anyway.

M's sleep notifier then alerts the sleep proxy S that M is going to sleep, providing a list of M's TCP ports in the *listening* state (actively listening for incoming connections). Assume that the SSH port, 22, is one such port.

Upon receiving the notification, S adds M to its list of proxied clients and sends out an ARP probe (Sec. 4.3.1), re-mapping the switched Ethernet to direct future packets for M to the network port at which S resides. S now begins receiving traffic that was meant for M. S responds to ARP requests and IPv6 *Neighbor Discovery* packets as if it were M, thereby maintaining M's network presence and ensuring traffic for M continues to arrive at S.

Some time later a remote client C attempts to connect to the sleeping machine M, using SSH. As the first transport-layer action taken in establishing this new connection, C sends a TCP SYN on port 22 to M which the switched Ethernet routes to S.

Upon examining the packet, S determines that it is a TCP SYN meant for M and destined to a port on which M was listening when it when to sleep. S therefore wakes M up by sending it a WOL packet (Sec. 3.3.3), removes M from the proxied client list, and drops the TCP SYN. As M wakes up, it sends its own ARP probes, which ensure that future traffic meant for M will arrive at M's network port. Meanwhile, C retransmits this SYN following the normal TCP timeout. The retransmitted SYN arrives at M, who responds as normal, thereby establishing the TCP connection without C being any the wiser, except for a small delay - quantified in Sec. 7.5.1.

4.2 The Sleep Notifier

Installing the sleep notifier on sleep clients greatly simplifies the overall design. As the service runs on user desktops, our aim is to make the sleep notifier code robust and stateless, requiring as simple configuration as possible.

The primary purpose of the sleep notifier is to notify the sleep proxy when the machine is going to sleep. Just before a machine is put to sleep, the Windows OS sends out a 'get ready for sleep" (a `Win32_PowerManagementEvent`) event to all the processes and drivers running on a machine, allowing them to prepare for sleep. The sleep notifier registers to receive this event. Upon receiving the event, the notifier immediately broadcasts a *sleep notification* packets (encapsulated in a UDP packet to port 9999), containing a "going-to-sleep" opcode and list of the sleep client's listening TCP ports, to the subnet broadcast address. For reliability it retransmits the packet three times.

In keeping with our light-weight approach, sleep notification packets are broadcast. The sleep client does not need to know the identity of the sleep proxy and requires no configuration nor stable storage, as there is no state to be kept. The sleep notification packet obviates the need for active probing sleep clients to determine sleep status (as done in [29]) or which ports should be proxied ([29] restricted proxied ports to a manually pre-configured set).

Since the sleep notifier may have less than two seconds in which to send the sleep notification packet before the machine falls asleep [6], it is possible, albeit unlikely, that the notification packets will not be sent in time. Consequently, the sleep notifier also sends out periodic heartbeats when the machine is awake. These heartbeats are identical to the sleep notification packet, save that they use a “heartbeat” opcode. In our current implementation, heartbeats are sent out every 5 minutes, with some jittering. When the sleep proxy misses two consecutive heartbeats from a client, it immediately sends a WOL packet to that client. If, after sending the WOL, neither a heartbeat nor a sleep notification is subsequently received from the client, the proxy assumes that the machine has left the network and removes it from the list of proxied sleep clients.

4.3 The Sleep Proxy

The sleep proxy needs to monitor incoming traffic to the sleep client and also wake that client by sending a WOL packet on the subnet broadcast address [7]. Redirecting traffic destined for a given machine to another machine outside of its local subnet requires substantial support from routers. Thus, the sleep proxy has to run either on the subnet router itself, or on some other subnet machine. Running a sleep proxy on the subnet router was not possible, so we use one dedicated machine per subnet to act as a sleep proxy for machines in those subnets.

4.3.1 Rerouting

Like most enterprise networks, our network is a switched Ethernet network. Thus, unicast traffic for a host is not generally visible to other hosts on the network. Thus, upon receiving the sleep notification from a client, the sleep proxy needs to ensure that the traffic destined for sleeping clients is re-routed to the sleep proxy’s NIC.

While there are a few ways to affect such re-routing, we have found sending *ARP probes* [13], as shown in Fig. 2, to be the most reliable method. A machine uses these ARP probes to advertise its MAC and IP address, and to perform duplicate address detection (DAD). Also, the subnet switches refresh/remap their internal routing tables upon receiving these probes.

Thus, when a sleep proxy receives a sleep notification from a client, it issues specially crafted ARP probes *pretending to be the sleep client* (refer again to Fig. 2).

[6]The sleep notifier cannot reliably force the system to remain awake once the notification is broadcast

[7]This packet must be broadcast since at the time it is sent, the subnet’s routing is set to deliver all packets meant for the sleeping host to the sleep proxy.

	Field	Value
Ethernet Header	Source Addr	$M.MAC\_Addr$
	Destination Addr	FF:FF:FF:FF:FF
ARP Request	Sender MAC Addr	$M.MAC\_Addr$
	Sender IP Addr	0.0.0.0
	Target MAC Addr	00:00:00:00:00:00
	Target IP Addr	$M.IP\_Addr$

Figure 2: ARP probe for sleep client M

This ensures that subsequent network traffic meant for the sleeping machine is delivered to the sleep proxy instead.[8]

When a sleeping machine wakes (either because the sleep proxy woke it, or because it was woken for some other reason), it will naturally send out a fresh set of ARP probes generated by the OS to ensure that it can re-use the same IP address that it had before it went to sleep. This has two nice side effects. First, the subnet switches now begin forwarding traffic meant for the sleeping (and now awake) machine, back to that machine, instead of the sleep proxy. Second, as these probes are broadcast, the sleep proxy can see them, allowing it to immediately recognize when clients have woken and cease proxying.

4.3.2 Reaction Policy

As discussed earlier our sleep proxy reaction policy responds to IP address resolution traffic, examines incoming TCP connection attempts, and ignores all other traffic. This means that (a) current TCP connections are broken and (b) UDP applications are not supported.

Intuitively, the former would seem to be a safe strategy for many applications. The sleep proxy is not responsible for putting a machine to sleep. That decision is taken by the local OS. If the local OS deemed it safe to put a machine to sleep while it had TCP connections open, then clearly the applications to which those TCP connections correspond have not placed requests to prevent sleep (a standard feature of modern OSes). Moreover, most common corporate network applications are inherently disconnection tolerant (e.g., email, web browser).

As for the latter, in our network, practically all desktop applications use TCP. Users typically access their machines either via SMB (to retrieve files) or via Remote Desktop. Upon initiation, both these applications start new TCP connections, and hence send corresponding SYNs. Routine maintenance is handled via RPC calls, and this traffic also goes over TCP. Additionally, it given the flexible parsing power of our sleep proxy, it should not be difficult to extend our technique to cover UDP traffic meant to initiate new connections for particular applications requiring such (e.g., NFS version 2).

The impact of ignoring non-TCP traffic and breaking currently existing TCP is difficult to estimate empirically.

[8]An alternate way of doing this would be to replace $M.MAC\_ADDR$ with the sleep proxy’s MAC address, however this could cause the DAD mechanism to be triggered if the sleep client were to wake very quickly after sleep.

However, we believe the proof is in the pudding: after months of running our code, none of our users or IT staff have complained that their machines did not wake on remote access and the only applications which we received request support for were the two cloud-based applications run by a small minority of users. [29] provides a more detailed discussion of our reaction policy and comparison with other possibilities.

4.4 Implementation Challenges

IPsec: Responding to IP address resolution traffic is easy: the sleep proxy simply issue ARP responses and Neighbor Discovery advertisements as if it were the sleeping client. Handling TCP connection attempts is more complicated. To detect an incoming TCP connection attempt the sleep proxy must examine the packet’s IP header confirming it was destined to a currently proxied machine, and contains a TCP SYN with a destination port on which that machine had been listening. While it is easy to parse a TCP SYN contained in a vanilla IPv4 or IPv6 packet, our network (like most corporate networks) is more complicated in both its use of IPv6 tunneling and IPsec ESP authentication[9]. Tunneling comes in three flavors, ISATAP, 6over4, and Teredo [36]. Our current implementation handles ISATAP and 6over4. ISATAP packets are already unwrapped for the sleep proxy by the ISATAP router and arrive as IPv6 packets on the sleep client’s ISATAP IPv6 address. Thus these packets require no additional processing. 6over4 packets arrive as IPv4 packets whose next protocol is 6over4. The inner packet is then removed and parsed as a standard IPv6 packet. Our current implementation does not handle Teredo wrapping, since it is being phased out in favor of the first two mechanisms.

The use of IPsec [42] presents a number of challenges. Imagine a remote machine C trying to connect to sleeping machine M using TCP. Let S be the sleep proxy. If IPsec is in use, there are two possibilities. Either C has not communicated with M in recent past, or it has.

If C has not recently communicated with M it would first try to establish a new security association by doing IPsec key exchange (IKE). The IKE packets are sent via UDP. The IKE sent by C end up at S. Recall, however, that our sleep proxy wakes up packets only on receiving TCP SYNs. Thus, the sleep proxy would never wake up M. However, Windows optimizes connection establishment by requiring C to send a TCP SYN “in the clear” as it begins the key exchange [42]. This is done to speed up the connection establishment: TCP handshake can happen in parallel with IPsec handshake. This works in our favor: the sleep proxy can detect the TCP SYN transmitted by C, and wake up M, which can then finish the key exchange. Otherwise, M would need to be woken for every IKE attempt. As we shall see later, in our network this would have lead to many spurious wake-ups.

[9]Note that tunneling and IPsec can be (and indeed are) used together. Our sleep proxy routinely sees and handles TCP SYNs that are encapsulated in an ESP payload, which is carried in an IPv6 packet, which is tunneled inside an IPv4 packet.

Conversely, if C has recently communicated with M, it may have cached the security association information. Since our network uses Encapsulated Security Payload (ESP) [25] protocol, C would encrypt the TCP SYN it sends. While the TCP SYN would end up at S, there is no way for S to decode the packet. This would have been incompatible with our reaction policy, except that our network uses ESP only with integrity service: the payload itself is not encrypted. Thus, S can parse the packet, inspect it, and wake M if needed.

Thus by choosing an IPsec setup in which both ESP payload encryption is disabled and enabling TCP connection establishment optimization, the need for running a heavier-weight reaction policy is ameliorated.

ARP probe timing: The sleep proxy cannot simply send out ARP probes as soon as it receives the sleep notification from a client, as that client may send other packets before the network card sleeps. If ARP probes from the sleep proxy intermingle with traffic generated from the client that is about to fall asleep, the spanning tree protocol may end up in state where packets meant for the sleeping machine are not routed to the sleep proxy. In our early implementations, this problem created much heartache.

To avoid this problem, after receiving the sleep notification, the sleep proxy begins pinging that sleep client. The sleep proxy waits for five consecutive ping failures before sending out ARP probes and thereby taking over for the sleeping client.

Daily wakeup & DHCP lease expiration: Currently, the sleep proxy wakes all sleeping clients at 5AM. The primary reason is to allow these machines to initiate any backup or scanning activity. The wakeup also obviates the need for the sleep proxy to handle DHCP traffic on behalf of the clients. In our network, DHCP leases are valid for 30 days. When the client is awake, it renews the lease every day. Furthermore, it also renews the lease when it wakes up. As each client is guaranteed to wake up at least once a day, we did not need to implement DHCP renewal on our sleep proxy. The same mechanism also protects against address black-holing: whereby a sleep proxy keeps holding on to the address of a machine that has departed the network. If heartbeats are not seen for a sleep client after the daily wakeup, that machine is inferred to have left the network (as described earlier).

Failure of sleep proxy: In our current implementation, each subnet is served by a single sleep proxy. This creates a single point of failure. We have designed, but not yet implemented a primary-backup solution for ensuring additional reliability. Another possibility is to design a purely peer-to-peer solution (Sec. 8). Our design does offer protection against a sleep proxy crashing, and restarting. The sleep proxy stores the MAC addresses of all the machines that it is proxying for in a log maintained on non-volatile network storage. Upon restarting, the sleep proxy checks the log, and proactively wakes up all the machines by sending them WOL packets. This ensures that the sleep proxy starts operations in a consistent state.

Multi-homed machines: The sleep proxy architecture can easily handle multi-homed machines as long as (i) the sleep notification goes out on all interfaces and (ii) a sleep proxy is available on each network that receives incoming connection attempts.

Manual wakeup: Apart from the "automatic" wakeup described so far, we also provide for remote, manual wakeup of sleeping clients. This is achieved by maintaining a website outside our corporate firewall. Every sleep proxy maintains an open TCP connection to this web server. Users can type in the name of their machine on this website. The web service sends the name to every sleep proxy, and if a sleep proxy has the specified machine as a client, it wakes that machine up by sending it a magic packet. This service provides a "last resort" wakeup alternative and also allowed the small minority of cloud application users to manually reconnect cloud apps.

5 Instrumentation

Our sleep proxy keeps a detailed log of its interactions with clients, including when and why the clients go to sleep or wake up. On client side, we use *Joulemeter*, to estimate the power consumption of the clients, and gather information about why clients *stay* awake. Joulemeter is installed as a separate, optional service on clients.

5.1 Monitoring power consumption

To quantify the energy savings of our approach, we desired an accurate method of estimating our deployment's power consumption. Different machine makes and models consume power at differing rates. Further, a given machine consumes vastly different power depending on its CPU utilization level, P-state and whether its monitors are on or off. For instance, the power usage of an HP xw4300 workstation with two monitors varied from 141W to 240W with processor utilization, and changed by an additional 120W with monitor power state for a total variation of 2.5X.

However, desktop workstations do not typically have built-in instrumentation to measure power usage, and we wished to avoid attaching external power-meters to each machine for the same reasons we rejected hardware augmented sleep proxying approaches. Consequently, we used a software solution, *Joulemeter*, that produces power usage estimates based on hardware activity and pre-calibrated machine models.

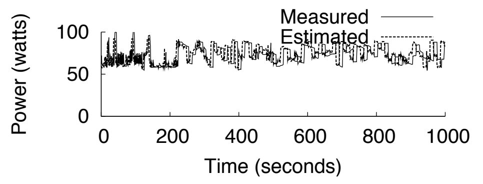


Figure 3: Measured and predicted power consumption

The key principle behind Joulemeter's energy estimation is to use a machine specific power model. The model consists of a set of equations that relate the hardware configuration and resource utilization levels to power usage. Our current model takes into account processor P-states, processor utilization, disk I/O levels, and whether the monitor(s) are on or off. The power model for a specific hardware configuration is learned via *calibration* - controlled experiments in a laboratory settings. Once the power model is known, the machine's power consumption at run time can be estimated by monitoring CPU utilization (and P-state), disk utilization and monitor status. We omit the details of model construction due to lack of space. For a preliminary introduction see [24].

Fig. 3 shows Joulemeter estimates versus measured power consumption (using a hardware power meter) for a HP d530 workstation with 2.66GHz Pentium CPU running a workload generator that loaded the CPU and disk at random. The estimates were generated using the calibrated model produced from *a different* workstation with the same model and CPU. The results shown confirm Joulemeter's estimates track closely with the actual power consumption. In practice, no two systems are exactly alike. Still, in validation testing we found Joulemeter predictions to be accurate within 20%

In our deployment Joulemeter generated power readings were averaged over 30 second intervals and periodically uploaded to the database. We have built up a library of power models covering most of our client machines.

5.2 Monitoring machine insomnia

To determine why a machine is awake, Joulemeter relies on two sources. First, it periodically checks the *lastUserInput* timer provided by the OS. This timer provides the time of last user activity. We compare the value of this timer to the idle timeout (a typical Windows default value is 30 minutes). If user activity has occurred more recently than the idle timeout, we assume that the machine is being kept awake by user activity. We note that due to various technical issues this timer is not always available, so we cannot always determine whether the user is active.

We also find that machines often stay awake even when the idle period exceeds this duration. To determine the reasons behind this, we rely on *powercfg.exe* utility that ships as part of Windows 7. The utility can often (but not always) shed light on why a machine is staying up by detailing *requests* to the OS for the machine to remain

awake. For example, a remote machine may be holding a file open or a defragmenting routine may be running. Joulemeter periodically collects this information and reports it to the central database. Analysis of this information is presented in Sec. 7.

6 Implementation and Deployment

Our deployment consisted of 6 proxies (one for each of our network's 6 wired subnets), 51 clients, an SQL database, and the manual wakeup webservice mentioned earlier (standard IIS webserver with code written using ASP.NET). Most of the code is written in C# (5000 lines).

Only the sleep proxy contains any significant amount of unmanaged code. The sleep proxy relies on PCAP to capture and examine incoming packets. A small custom driver allows the sleep proxy to craft and inject ARP probes while bypassing the network stack. The primary data structure in the sleep proxy is a hashtable used to keep track of clients and their status. We first used ordinary desktop machines as proxies and have begun migrating to the low-powered, small-form-factor machines drawing less than 25 watts of power.

On client side, apart from the required sleep notifier service, the clients install three optional applications: Joulemeter, a GUI program displaying sleep statistics and estimated energy savings, and an auto-updater service that keeps client-side code up-to-date. During client installation, we ensured that Wake-On-LAN was enabled and ARP offload (which is enabled by default for certain cards in Windows 7) was disabled on the client's NIC. We also set the idle timeout to 30 minutes.

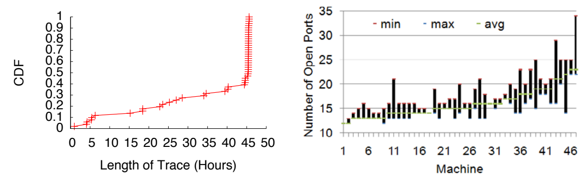
7 Results

This section is guided by several overarching questions. What is the sleep and wake behavior of machines in our system? How much power did our solution save? What might be done to obtain additional power savings? What impact did our setup have on user experience? Was the sleep proxy architecture scalable? For the impatient reader, we highlight our main insights at this section's end (Sec. 7.6).

We begin by describing the details of our dataset.

7.1 Dataset Overview

While our deployment has been active for half a year in various stages, for the rest of this section we focus on the 45 day period from November 19th, 2009 through January 3, 2010. During this time, we gathered data from 51 distinct machines belonging to 50 distinct users. As users installed our software at differing times, not all machines provided data for the entire period (although most did). Fig. 4(a) shows the cumulative distribution of trace



(a) CDF of Machine Trace Length (b) Number of Listening Ports

Figure 4: Trace Length and Listening Port Distribution

lengths of individual machines. Our users were a self selecting group, so their behavior may not be representative of all user populations.

7.1.1 Machines in our study

As we noted in Sec. 5.1, machine power consumption depends on the particulars of that machine's hardware configuration. The hardware configuration of machines in our deployment was varied, but not overly so. Of the 51 machines, 43 are HP and 6 were Dell. Only one of the machines has an AMD processor, the rest having Intel CPUs. Most of the machines are dual or quad cored. The CPU frequencies vary from 2-3.4GHz. Twenty seven machines had one monitor, 20 had two, and five had three. Five machines ran Windows Vista, all the rest ran Windows 7.

As we wake up machines for incoming TCP SYNs only on listening ports, it is worth examining the number of listening TCP ports on each machine. This number, of course, varies over time, as active processes and settings change. Fig. 4(b) shows the min, max, and average number of listening ports by machine. One machine had 35 ports open simultaneously!

7.1.2 Traffic

Since all traffic destined for sleeping clients arrives at their sleep proxies, we can examine this traffic in centralized manner, without installing sniffers on individual machines. While we have deployed a sleep proxy on each of our six subnets, 59% of our machines are connected to the largest subnet. We have seen as many as 800 active machines on this subnet. We examined in detail a trace of all (23 million) packets arriving at the sleep proxy serving this subnet during a typical work week (5.5 days).

Of this traffic, 96% were multicast and broadcast packets. Of the multicast packets, 12.31% were ARP requests, which the sleep proxy examined and replied to as needed. The vast majority of the multicast traffic was safely ignorable [29]. The remaining 4% traffic was unicast: destined either to the proxy itself, or to the sleeping clients. 75% of these packets were wrapped by ESP and 8.4% were tunneled v6-over-v4 packets - underscoring the importance of parsing such packets. 7% of the total unicast packets were UDP (mostly IPsec related) and 3% were ICMP, which the sleep proxy ignores. Most of the remaining traffic was TCP, and the proxy was able to ignore the vast majority of it. During this time, we woke sleeping clients for just 747 TCP SYNs. Our analysis of the traffic data confirmed the importance of filtering TCP SYNs based on port. More than half of incoming TCP connection attempts were destined to ports on which the sleep client was not listening. If we had woken clients without filtering by port, we would have had more spurious wake-ups than valid ones.

7.2 Sleep/Wake Behavior

We note that five of our 51 clients did not sleep at all, as their their users manually disabled sleep functionality.

7.2.1 Aggregate sleep/wake behavior

Fig. 5(a) shows the percentage of time each machine spent sleeping, as a CDF across all machines. The uniform slope of the CDF demonstrates that the average sleep time was quite variable, with 50% of the clients sleeping more than half the time. Fig. 5(b) plots the CDF of the average number of sleep-to-wake transitions per day for the machines. Most machines average fewer than seven daily wake-ups. Later, we will see that most of these wake-ups were caused by IT management traffic (e.g., updates) arriving for a sleeping machine.

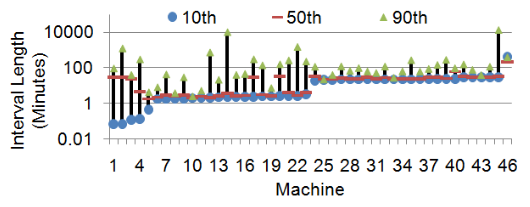
We now examine the duration of sleep and awake intervals. Note that no sleep interval is longer than 1440 minutes because of the daily 5AM wakeup. The CDF of length of sleep and wake intervals is shown in Fig. 5(c), while Fig. 5(d) shows the time-weighted CDF (i.e., contribution of intervals at or below a given length to the total sleep or wake time). By comparing these two figures, we see that while most sleep and awake intervals are under one hour, the majority of both sleep and awake time comprises intervals over one hour. This implies that insomnia should be our first focus in attempting to reduce power usage (Sec. 7.3.2).

The awake interval CDF in Fig. 5(c) demonstrates a bi-modal distribution with abrupt changes in slope at around two minutes, and at 30 minutes. This indicates that awake periods of two and 30 minutes are prevalent in our trace.

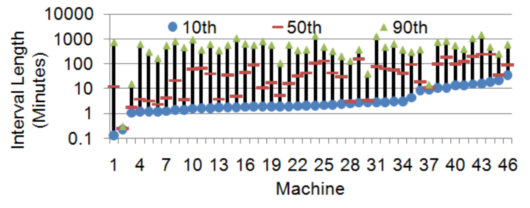
7.2.2 Individual sleep/wake behavior

Figs. 6(b) and 6(a) show the 10th, 50th, and 90th percentile of wake and sleep intervals for each machine. The machines are sorted in order of 10th percentile. Notably, for around half of the machines the 10th percentile lies around two minutes, while for other half it lies around 30 minutes, corresponding to the jumps seen in Fig. 5(c).

We closely inspected a number of these awake periods. The prevalence of both two and 30 minute awake periods is easily understood: these being the idle timeouts after WOL wakeup and user activity respectively. When looking at our special 5AM wakeup (which we know was not user-initiated - Sec. 4.4) we saw a much greater than normal proportion of two minute wakes which is precisely what we would expect.

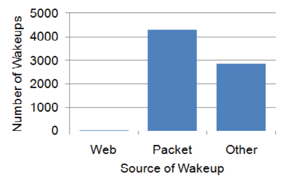


(a) Awake Interval

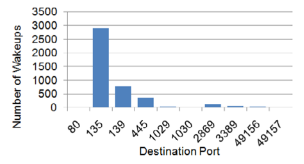


(b) Sleep Interval

Figure 6: Per-machine Sleep/Awake Intervals



(a) Cause of Wake



(b) Wakeup Source Port

Figure 7: Cause of wake-ups

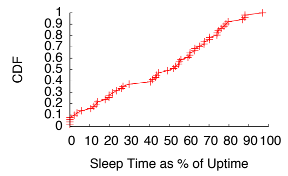
Fig. 6(b) shows that for about a quarter of the machines, the median sleep interval is under 10 minutes. For one machine all sleep intervals were under a minute. This machine appears to have some driver configuration issue that causes almost immediate wake upon sleep and was unique in our data set. Such intervals add very little to overall sleep duration and indicate potential sleep problems which will be examined further in Sec. 7.3.

7.2.3 Why do machines wake up?

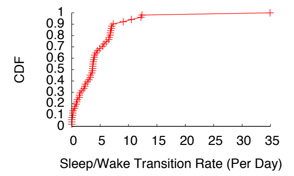
Fig. 7(a) shows the causes of wake-ups. We divide these into three categories: manual wake-up using our web site, wake-up by proxy due to incoming traffic, and other. The last bucket includes wake-ups caused by users walking up to the machine, any timer-based wake-ups caused by the BIOS, as well as occasional WOL packets sent by a commercial wakeup solution being tested by our IT department. We were able to confirm for 33% of these that the user did in fact initiate wakeup (by checking *lastUserInput* - Sec. 5.2) and for 50% of these the user definitively did not wake the machine. The remaining 27% could not be determined as *lastUserInput* was unavailable.

We see that while the web site was used in a few cases, it is not statistically significant. The majority of wake-ups caused by the sleep proxy are due to incoming TCP SYNs. The ports to which these SYNs were destined to are shown in Fig. 7(b).

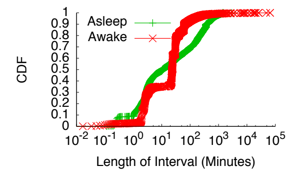
Remote Procedure Calls (port 135) were the overwhelmingly largest source of wakeup triggers, followed



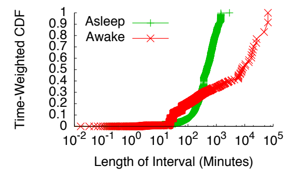
(a) Time Sleeping(%)



(b) Wake Transitions per Day



(c) CDF of Sleep/Awake Intervals



(d) Time-Weighted CDF

Figure 5: Aggregate Sleep/Wake statistics

by NETBIOS (139) and SMB (445). SMB is the main mechanism used for remote file system access in our network. The two other notable ports are UPnP (2869) and Remote Desktop (3389). In our network, Remote Desktop is the primary mechanism for interactive remote machine access. We can see Remote Desktop is not a major wakeup source. In fact, only 39% of the machines were ever woken up due to Remote Desktop requests. Therefore, it would seem that while users leave their machines on for potential remote access, interactive remote access is used relatively rarely.

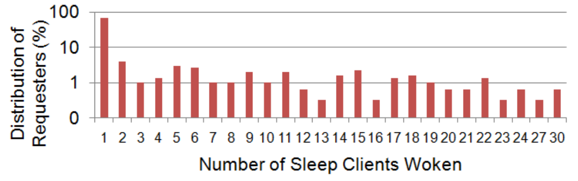
7.2.4 Who wakes up machines?

There were slightly over 300 IP addresses *requesters* whose incoming connection attempts caused wake-ups. Most of these only attempted to connect to a single sleep client. However, a sizable minority attempted to connect to multiple clients as seen in Fig. 8(a). We were able to verify that all the requesters who woke 20 or more sleep clients were machines belonging to our IT department. These machines perform a variety of management actions such as verifying patch status and checking security policies. We will see later that our IT configuration is sleep-unfriendly in other ways as well (Sec. 7.3.2).

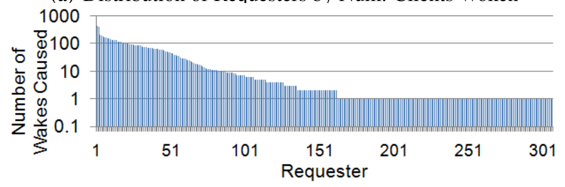
Fig. 8(b) shows the number of wakeup events caused by requester. Just as most requesters only connect to a single machine, many only cause only one wakeup and most cause only a handful. However, again a large minority of requesters cause many wake-ups each. IT-owned machines again make a large portion of this group. Interestingly, several of the most active requesters actually connect to only one or a handful of machines. In fact, the most active requester with over 400 requests connected to only two machines, and that too in in a span of just two weeks! We are currently investigating the role of this requester further.

7.3 Why Machines Don't Sleep Better

While we have seen that our solution is fairly successful at enabling machines to sleep (Fig. 5(a)), we wanted to investigate whether more idle time could be harvested. We begin by noting that most machines are not being woken overly often (Fig. 5(b)). However, a small subset of machines suffer from "crying-baby-syndrome" being woken as soon as they fall asleep. Sec. 7.2.4. These machines are



(a) Distribution of Requesters by Num. Clients Woken



(b) Number of Wake-ups Caused By Requester

Figure 8: Who causes wake-ups?

being bombarded by frequent connection attempts that interrupt their sleep often. If a machine with a standard 30 minute idle timeout wakes 12 times a day, one quarter of the day will have been spent awake due to wake-ups alone. It appears that configuration issues are responsible for much of this behavior.

However, most sleep clients are being kept awake for other reasons the majority of the time. In fact, when not being kept awake, these machines manage to sleep well, sustaining few wakeup events per day. We now consider whether these machines would have benefited from a more aggressive idle timeout, and then look at the problem of insomniac machines.

7.3.1 Aggressive idle timeout

As mentioned in Sec. 7.2.2, it appears that setting the idle timeout more aggressively could result in some power savings. We now consider how much could be saved with a 5-minute idle timeout (this is 1/3rd the EnergyStar guidelines recommendation [16]).

To do so, we examined each wake interval to see why the machine was being kept up. Recall from Sec. 5.2 that a machine may be kept awake because the user is active, the machine has woken up recently, or a *stay-awake* request placed by a local application with the OS.

We divided the total awake time into three components, *recoverable*, *unknown*, and *unrecoverable*. *Recoverable* time was time in which the machine could have slept if the idle timeout had been set more aggressively. This time was the sum of periods in which the user had been active

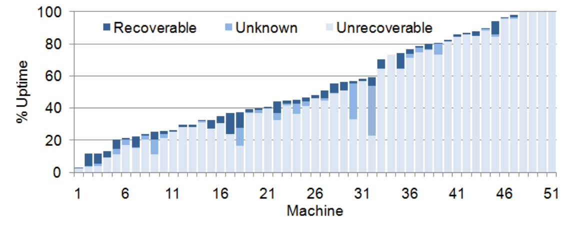
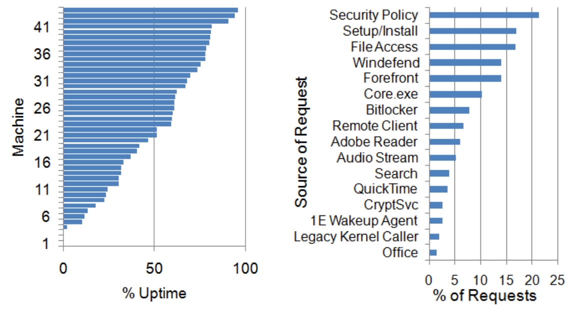


Figure 9: Awake Time as Percentage of Uptime. Broken Into Components Unknown, Recoverable, and Unrecoverable using Aggressive Idle Timeout



(a) Percentage of Awake Time Caused by Requests

(b) Source of Requests

Figure 10: Stay-Awake Request Data

within the past five minutes or the machine had been woken within the past five minutes. The *unknown* time was the time for which insufficient data was available to diagnose cause of wakefulness. The *unrecoverable* time consisted of all other time (i.e., an application had placed a *stay-awake* request with the OS).

Thus the recoverable time is a lower bound on the awake time that could have been saved by setting a more aggressive idle timeout. The sum of recoverable and unknown time provides the upper bound. Fig. 9 breaks the total wake time as percentage of uptime into these three components on a per-machine basis. We see that on most machines the impact would be relatively small. These machines are being kept up by local application stay-awake requests, to which we now turn.

7.3.2 Insomnia

We now look more closely at which local applications keep machines awake. We label this phenomenon *insomnia*. Fig. 10(a) plots the fraction of awake time a given machine was kept awake by local applications requesting OS to prevent sleep. We see that the majority of awake time is in fact due to such stay-awake requests. So which applications cause these stay-awake requests? Fig. 10(b) shows the percentage of requests initiated by various applications. The news here is heartening. Four of the top sources (Security Policy Agent, Windefend, Forefront, and Bitlocker) are all applications mandated by our IT department. It may be possible to reconfigure or even re-write these applications to minimize and coordinate the duration of time they are active (and thus preventing sleep). At least three more (Flash, Quicktime and Audio Stream) are the result of code or driver bugs. For example, certain older versions of Flash player may keep a machine awake by playing silence even after the audio clip has finished (Windows prevents sleep when audio streams are active). The third-highest request source is SMB. SMB's default behavior prevents a machine whose files are being accessed from sleeping. Careful changes to this behavior may allow for greater sleep opportunities.

7.4 Power savings

PC consumption varied, averaging from 89-143W for individual machines. The lowest draw we saw was 50W idling. The highest was 191W heavily loaded. While sleeping, all machines drew 1-2W. Monitors generally added from 30-60W when on.

Fig. 11(a) illustrates the lower bound on power savings on a per machine basis. This lower bound is calculated with the assumption that had the machine stayed up instead of sleeping, it would have consumed power at the lowest rate seen in the entire non-sleeping portion of the trace. This represents part of the reason we saw less power savings than that predicted by previous work (which assumed machines consumed power at a constant rate irrespective of activity level). The average across all machines is about 20%, although variation is considerable.

Fig. 11(b), shows aggregate power consumption for a both a representative one-week period beginning 12/3/09 and the winter break (beginning 12/24/09). During the representative week, weekend power consumption is low, spiking only at the 5AM wakeup. During the work-week, power use peaks during the work day before declining into an overnight trough and bottoms out early on Friday. In contrast we can see a markedly different pattern for the Mid-Winter week with almost no increase in activity during the day from the day preceding Christmas (which fell on Friday) through the following Monday. By the Tuesday following the holiday, we begin to see a similar level of activity to that of the representative week, albeit at a lower amplitude, as employees begin returning from the holiday. Interestingly, the power consumption over the Christmas weekend (12/26-12/27) weekend was slightly higher than during a normal weekend (12/5-12/6).

7.5 Micro-Benchmarks

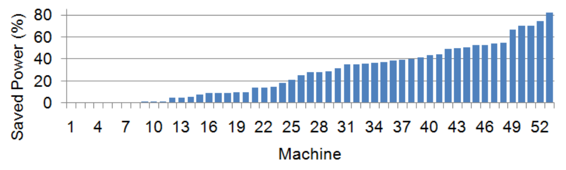
We now validate our architectural approach by examining wakeup delay time and sleep proxy scalability.

7.5.1 Wakeup delay

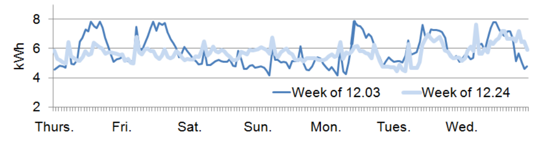
The energy saved by our system comes at a cost: the user experiences additional *startup* latency *the first time* a connection (e.g., ssh login or samba file access) to a sleep

Step	Time (s)	From→To	Packet Type
1	0	M1→M2	TCP SYN
2	0.04	S1→Broadcast	Magic Packet
3	2.48	M1→M2	TCP SYN
4	5.6	M2→Broadcast	ARP Probe
5	8.48	M1→M2	TCP SYN
6	8.49	M2→M1	TCP SYN-ACK

Table 1: Time line of a wakeup



(a) Lower Bound on Per-Machine Power Savings



(b) Aggregate Power Draw for Normal vs. Mid-Winter Weeks

Figure 11: Power Draw and Savings

client is attempted since that client fell asleep. This happens because sleep client takes time to both wake and begin responding to an incoming TCP connection attempt. To make the system usable, we need to minimize the startup latency encountered by interactive transactions.

The user-perceived startup latency consists of several components: the delay involved in sending the WOL magic packet, the time required to wake up the machine, and the time required to perform any application-specific actions. To quantify these component latencies, we present a simple, but representative example.

Two machines, M1 and M2 were connected to the same subnet. M1 was ran a simple TCP sink, and was put to sleep. Thereafter, sleep proxy S1 started proxying for M1. From M2, we attempted to establish a TCP connection to the the sink on M1. The packet trace of the connection establishment is summarized in Table 1.

The total latency is about 8.5 seconds, but the sleep proxy itself consumes only 40 milliseconds, even though it is on a busy subnet and proxying for several other machines. The largest component is the *wake-up* delay (i.e., time required for M2 to wake up and become active). This is roughly the delay between steps 2 and 4 (about 5.5 seconds). The remaining *TCP-retransmit* delay occurs between steps 4 and 5 (about 3 seconds). This delay is incurred while M1 waits to retransmit the TCP SYN the second time, following regular TCP timeout algorithm [31].

Specific applications will usually encounter slightly higher latencies, as the machine needs to perform additional, application-specific actions. For example, when M1 tried to list a directory on M2 via SMB, the transaction took 13.37 seconds when M2 was asleep. The additional delay was incurred while M2 re-connected with the domain controller, and obtained security credentials to determine whether to allow M1 access.

We stress that this delay is incurred only for the transaction that wakes the machine. Subsequent transactions experience normal latencies. While our experience is that users do not mind this one-off penalty, both the wake-up and retransmit delays can be addressed. A number of research and engineering efforts are underway to address the former. The latter can be shortened either by having M1 retransmit TCP SYN more aggressively, or having S1 "replay" the TCP SYN.

7.5.2 Scalability

Our current deployment uses one sleep proxy per subnet. The load on these sleep proxies is a potential concern. We find that the CPU load on a sleep server rarely exceeds 5%. The total traffic (broadcast inclusive) seen by the sleep server is also quite low (90th percentile is 250Kbps). We conclude sleep proxy operations do not require substantial resources, and a single sleep proxy could easily handle very large subnets if necessary. Conversely for reasonably sized subnets, the sleep proxy could be located on a client machine without noticeably degrading the user experience (Sec. 8).

7.6 Summary

Insomnia is the foremost cause of lost sleep. Thus improving the energy savings of systems like ours, the main focus should be on addressing sources of wakefulness.

IT applications are the main source of both insomnia and fitful sleeping. Several uncoordinated IT applications for patching, security, and network testing all woke machines and kept them awake. While we studied one particular IT setup, practically all IT setups will interfere with sleep to some extent - dependent on quantity, aggressiveness and degree of coordination of IT applications.

Misconfiguration can result in crying-baby syndrome Requiring administrators to diagnose and resolve the minority of machines suffering this issue.

Use of more aggressive idle timeouts is of secondary benefit. In enterprise systems behind firewalls, wakeups will occur because of valid incoming TCP connection attempts and in well configured setups, the number of wakeups caused by IT/misconfiguration will be minimal. Thus savings from more aggressive idle timeouts will be minor.

Incoming TCP connection attempts need to be filtered by listening port. More incoming TCP connection attempts arrived for non-listening ports, than listening ones.

8 Conclusion & Future Work

We have designed and deployed a light-weight network-based sleep proxy in an operation enterprise network on over 50 user workstations - the first such deployment of which we are aware. During our work, we uncovered and addressed several practical issues that must be addressed by light-weight sleep proxying systems in enterprise networks. Our system has functioned both to user satisfaction and our own specification for the past several months, providing significant sleep opportunities and power savings using a simple reaction policy. However, we find that significantly more power savings could be achieved by altering the IT setup. Additionally, certain classes of cloud applications require specialized reaction policies. Should use of such persistent cloud applications become more widespread, our reaction policy would need adjustment.

We conclude with a brief discussion of future possibilities and concerns.

IT application coordination and configuration: Currently IT maintenance tasks are uncoordinated and consequently will keep machines awake during each of their separate execution time periods. Devising methodologies that schedule these tasks to overlap as much as possible can significantly increase sleep opportunities.

P2P sleep proxy: Our current setup requires the use of a dedicated (albeit low-power) sleep proxy machine on each subnet. We are working on a p2p architecture in which machines fall asleep one after the other, while the "last man standing" keeps watch for the entire subnet.

Security: While the sleep proxying system does not pose a traditional security concern, we do note that many machines waking simultaneously could cause significant power spikes. To reduce the risk of this being exploited by an attacker, proxies can rate limit WOL packets sent.

References

[1] Advanced Configuration and Power Interfae, revison 4.0. http://www.acpi.info/.

[2] Adaptive Technologies. http://www.adaptiva.com/.

[3] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *NSDI'09*, Berkeley, CA, USA, 2009.

[4] 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*, 2010.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

[6] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future internet through selectively connected end systems. In *Hotnets*. ACM SIGCOMM, Nov. 2007.

[7] Intel Active Management Technology (AMT). http://www.intel.com/technology/platform-technology/intel-amt/.

[8] Apple Wake On Lan. http://www.macworld.com/article/142468/2009/08/wake_on_demand.html.

[9] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007.

[10] W. L. Bircher and L. K. John. Complete system power estimation: A trickle-down approach based on performance events. In *ISPASS*, 2007.

[11] J. Blackburn and K. Christensen. A simulation study of a new green bittorrent. In *Workshop on Green Communications*. IEEE Internation Conference on Communications, June 2009.

[12] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. Power awareness in network design and routing. In *INFOCOM 2008*, 2008.

[13] S. Cheshire. IPv4 Address conflict detection. RFC 4227.

[14] K. Christensen, P. Gunaratne, B. Nordman, and A. George. The next frontier for communications networks: Power management. *Computer Communications*, 27(18):1758–1770, Dec. 2004.

[15] K. J. Christensen and F. B. Gulledge. Enabling power management for network-attached computers. *Int. J. Netw. Manag.*, 8(2):120–130, 1998.

[16] T.-T. Committee. www.ecma-international.org/publications/files/drafts/tc32-tg21-2009-150.doc. Technical report, ECMA, Nov. 2009.

[17] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. G. Shin. Litegreen: Saving energy in networked desktops using virtualization. In *USENIX ATC*, 2010.

[18] Worldwide pc market. http://www.tgdaily.com/slideshows/index.php?s=200903062p=1.

[19] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2007.

[20] Gumstix. www.gumstix.com.

[21] M. Gupta and S. Singh. Greening of the internet. In *SIGCOMM*, New York, NY, USA, 2003.

[22] Idc netbook and pc sales projections. http://www.tgdaily.com/slideshows/index.php?s=200903062p=1.

[23] M. Jimeno, K. Christensen, and B. Nordman. A network connection proxy to enable hosts to sleep and save energy. In *Performance Computing and Communications Conference*, pages 101–110. IEEE, Dec. 2008.

[24] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics08*, June 2008.

[25] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC4301.

[26] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan. Energy aware network operations. In *Global Internet Symposium*. IEEE, April 2009.

[27] M. Maisto. Global pc market suffering first decline since dot com crash, September 2009.

[28] Adding and Deleting Wake on LAN Patterns. http://msdn.microsoft.com/en-us/library/ff543710.aspx.

[29] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: Reducing energy waste in networked systems. In *NSDI*, April 2009.

[30] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI'08*, Berkeley, CA, USA, 2008.

[31] J. Postel. Tranmission control protocol. RFC 793.

[32] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. In *HotPower'08*, 2008.

[33] Intel Remote Wakeup Technology (RWT). http://www.intel.com/technology/chipset/remotewake-qa.htm?iid=Tech_remotewake+qa.

[34] A. Sinha and A. P. Chandrakasan. Energy efficient real-time scheduling. *Computer-Aided Design, International Conference on*, 0:458, 2001.

[35] D. C. Snowdon, E. Le Sueur, S. M. Petters, and G. Heiser. Koala: a platform for os-level power management. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 289–302, 2009.

[36] Teredo tunneling. http://en.wikipedia.org/wiki/Teredo_tunneling.

[37] Tickless kernel. http://www.lesswatts.org/projects/tickless/.

[38] V. Valancius, N. Laoutaris, L. Massoulie, C. Diot, and P. Rodriguez. Greening the internet w/ nano data centers. In *CoNEXT*. ACM, 2009.

[39] Verdiem Technologies. http://www.verdiem.com/.

[40] D. Washburn. How much money are your idle pc wasting. Forrester Researech Reports, December 2008.

[41] C. A. Webber, J. A. Roberson, M. C. McWhinney, R. E. Brown, M. J. Pinckard, and J. F. Busch. After-hours power status of office equipment in the usa. *Energy*, Nov. 2006.

[42] How IPSec Works. http://technet.microsoft.com/en-us/library/cc759130.aspx.

[43] Wake-on-LAN. http://en.wikipedia.org/wiki/Wake-on-LAN.