USENIX

Proceedings of NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA March 30–April 1, 2011

conference

proceedings

NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA

March 30–April 1, 2011

Sponsored by

USENIX

in cooperation with ACM SIGCOMM and ACM SIGOPS

© 2011 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. Permission is granted to print, primarily for one person's exclusive use, a single copy of these Proceedings. USENIX acknowledges all trademarks herein.

ISBN 978-931971-84-3

USENIX Association

Proceedings of NSDI ’11:
8th USENIX Symposium on Networked Systems Design and Implementation

March 30–April 1, 2011
Boston, MA, USA

Conference Organizers

Program Co-Chairs

David G. Andersen, *Carnegie Mellon University*
Sylvia Ratnasamy, *Intel Labs Berkeley*

Program Committee

Aditya Akella, *University of Wisconsin—Madison*
Katerina Argyraki, *École Polytechnique Fédérale de Lausanne (EPFL)*
Remzi Arpaci-Dusseau, *University of Wisconsin—Madison*
Hari Balakrishnan, *Massachusetts Institute of Technology*
Andrew Birrell, *Microsoft Research*
Byung-Gon Chun, *Intel Labs Berkeley*
Jason Flinn, *University of Michigan*
Rodrigo Fonseca, *Brown University*
Paul Francis, *Max Planck Institute for Software Systems*
Brad Karp, *University College London*
Dina Katabi, *Massachusetts Institute of Technology*
Eddie Kohler, *University of California, Los Angeles, and Meraki*
Jinyang Li, *New York University*
Bruce Maggs, *Duke University and Akamai Technologies*
Ratul Mahajan, *Microsoft Research*
David Maltz, *Microsoft Research*
David Mazières, *Stanford University*
Jitendra Padhye, *Microsoft Research*
KyoungSoo Park, *KAIST (Korea Advanced Institute of Science and Technology)*
Jennifer Rexford, *Princeton University*
Alex C. Snoeren, *University of California, San Diego*
Lakshminarayanan Subramanian, *New York University*
Helen Wang, *Microsoft Research*
Bill Weihl, *Google*

Poster Session Chair

Michael Walfish, *The University of Texas at Austin*

Steeering Committee

Thomas Anderson, *University of Washington*
Brian Noble, *University of Michigan*
Jennifer Rexford, *Princeton University*
Mike Schroeder, *Microsoft Research*
Chandu Thekkath, *Microsoft Research*
Amin Vahdat, *University of California, San Diego*
Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Sharad Agarwal
Shuchi Chawla
Weidong Cui
Chuanxiong Guo
Dan Halperin
Srikanth Kandula
Aman Kansal
Changhoon Kim
Katrina LaCurts
Julio Lopez
Jay Lorch
David Molnar
Radhika Niranjan Mysore
Calvin Newport
Evdokia Nikolova
Bryan Parno
Milo Polte
Lucian Popa
Raluca Ada Popa
Russell Power
Bodhi Priyantha
Shravan Rayanchu
Michael Schapira
Sayandeep Sen
Deian Stefan
Martin Suchara
Cedric Westphall
Keith Winstein
Alec Wolman
Ming Zhang

NSDI '11: 8th USENIX Symposium on Networked Systems Design and Implementation
March 30–April 1, 2011
Boston, MA, USA

Message from the Program Co-Chairs. vii

Wednesday, March 30

Speed, Speed, and More Speed

SSLShader: Cheap SSL Acceleration with Commodity Processors 1
Keon Jang and Sangjin Han, KAIST; Seungyeop Han, University of Washington; Sue Moon and Kyoungsoo Park, KAIST

ServerSwitch: A Programmable and High Performance Platform for Data Center Networks 15
Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang, Microsoft Research Asia

TritonSort: A Balanced Large-Scale Sorting System 29
Alexander Rasmussen, George Porter, and Michael Conley, University of California, San Diego; Harsha V. Madhyastha, University of California, Riverside; Radhika Niranjan Mysore, University of California, San Diego; Alexander Pucher, Vienna University of Technology; Amin Vahdat, University of California, San Diego

Performance Diagnosis

Diagnosing Performance Changes by Comparing Request Flows 43
Raja R. Sambasivan, Carnegie Mellon University; Alice X. Zheng, Microsoft Research; Michael De Rosa, Google; Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger, Carnegie Mellon University

Profiling Network Performance for Multi-tier Data Center Applications. 57
Minlan Yu, Princeton University; Albert Greenberg and Dave Maltz, Microsoft; Jennifer Rexford, Princeton University; Lihua Yuan, Srikanth Kandula, and Changhoon Kim, Microsoft

Nothing but Net

Efficiently Measuring Bandwidth at All Time Scales 71
Frank Uyeda, University of California, San Diego; Luca Foschini, University of California, Santa Barbara; Fred Baker, Cisco; Subhash Suri, University of California, Santa Barbara; George Varghese, University of California, San Diego

ETTM: A Scalable Fault Tolerant Network Manager 85
Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy, University of Washington

Design, Implementation and Evaluation of Congestion Control for Multipath TCP. 99
Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley, University College London

Wednesday, March 30 (continued)

Data-Intensive Computing

CIEL: A Universal Execution Engine for Distributed Data-Flow Computing 113
Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand, University of Cambridge Computer Laboratory

A Semantic Framework for Data Analysis in Networked Systems 127
Arun Viswanathan, University of Southern California Information Sciences Institute; Alefiya Hussain, University of Southern California Information Sciences Institute and Sparta Inc.; Jelena Mirkovic, University of Southern California Information Sciences Institute; Stephen Schwab, Sparta Inc.; John Wroclawski, University of Southern California Information Sciences Institute

Paxos Replicated State Machines as the Basis of a High-Performance Data Store 141
William J. Bolosky, Microsoft Research; Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li, Microsoft

Thursday, March 31

Security and Privacy

Bootstrapping Accountability in the Internet We Have. 155
Ang Li, Xin Liu, and Xiaowei Yang, Duke University

Privad: Practical Privacy in Online Advertising 169
Saikat Guha, Microsoft Research India; Bin Cheng and Paul Francis, MPI-SWS

Bazaar: Strengthening User Reputations in Online Marketplaces 183
Ansley Post, MPI-SWS and Rice University; Vijit Shah and Alan Mislove, Northeastern University

Energy and Storage

Dewdrop: An Energy-Aware Runtime for Computational RFID 197
Michael Buettner, University of Washington; Benjamin Greenstein, Intel Labs Seattle; David Wetherall, University of Washington and Intel Labs Seattle

SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy 211
Anirudh Badam and Vivek S. Pai, Princeton University

Debugging and Correctness

Model Checking a Networked System Without the Network 225
Rachid Guerraoui and Maysam Yabandeh, EPFL

FATE and DESTINI: A Framework for Cloud Recovery Testing 239
Haryadi S. Gunawi, University of California, Berkeley; Thanh Do, University of Wisconsin, Madison; Pallavi Joshi, Peter Alvaro, and Joseph M. Hellerstein, University of California, Berkeley; Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison; Koushik Sen, University of California, Berkeley; Dhruba Borthakur, Facebook

SliceTime: A Platform for Scalable and Accurate Network Emulation 253
Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle, RWTH Aachen University

Thursday, March 31 (continued)

Mobile Wireless

Accurate, Low-Energy Trajectory Mapping for Mobile Devices 267
Arvind Thiagarajan, Lenin Ravindranath, Hari Balakrishnan, Samuel Madden, and Lewis Girod, MIT Computer Science and Artificial Intelligence Laboratory

Improving Wireless Network Performance Using Sensor Hints 281
Lenin Ravindranath, Calvin Newport, Hari Balakrishnan, and Samuel Madden, MIT Computer Science and Artificial Intelligence Laboratory

Friday, April 1

Datacenters Learning to Share

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center 295
Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica, University of California, Berkeley

Sharing the Data Center Network 309
Alan Shieh, Microsoft Research and Cornell University; Srikanth Kandula, Microsoft Research; Albert Greenberg and Changhoon Kim, Windows Azure; Bikas Saha, Microsoft Bing

Dominant Resource Fairness: Fair Allocation of Multiple Resource Types 323
Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica, University of California, Berkeley

Wireless and More

PIE in the Sky: Online Passive Interference Estimation for Enterprise WLANs 337
Vivek Shrivastava, Shravan Rayanchu, and Suman Banerjee, University of Wisconsin—Madison; Konstantina Papagiannaki, Intel Labs, Pittsburgh

SpecNet: Spectrum Sensing Sans Frontières 351
Anand Padmanabha Iyer, Krishna Chintalapudi, Vishnu Navda, Ramachandran Ramjee, and Venkata N. Padmanabhan, Microsoft Research India; Chandra R. Murthy, Indian Institute of Science

Towards Street-Level Client-Independent IP Geolocation. 365
Yong Wang, UESTC and Northwestern University; Daniel Burgener, Marcel Flores, and Aleksandar Kuzmanovic, Northwestern University; Cheng Huang, Microsoft Research

Message from the Program Co-Chairs

NSDI in 2011 carries on the conference's tradition of presenting the very best work in the area of networked systems. As in previous years, we take a broad view of the NSDI charter, selecting papers from across the range of the USENIX, SIGCOMM, and SIGOPS communities, rather than their intersection. The result is an exciting program with papers spanning topics from high-performance systems to security and privacy, from mobile wireless systems to tools for testing and model checking.

We received 157 paper submissions. All submissions that met the formatting and basic quality standards (three did not) were reviewed by several members of the program committee, and in a small number of cases we used external reviewers to complement the expertise of the program committee. The 26 members of the program committee completed 700 reviews. Each paper received at least 3 reviews; on average, the committee completed 4.5 reviews per paper, with some papers receiving as many as 9 reviews. These written reviews laid the groundwork for the program committee meeting in Berkeley, California, on December 3, 2010. All 26 members of the program committee attended the meeting to weigh the 56 papers selected for discussion. Papers selected for discussion had received an average of 5.9 reviews, and the meeting led to our final program of 27 papers. Because of the special role conferences play in our field, all papers were shepherded by a program committee member.

We are grateful to everyone whose hard work made this conference possible. Most of all, we are indebted to all of the authors who submitted their work to this conference. We thank the program committee for their dedication and hard work in reviewing papers and participating in the extensive discussions at the PC meeting, as well as in shepherding the final versions. We thank our external reviewers for lending their expertise on short notice. We extend special thanks to ICSI for hosting the program committee meeting in Berkeley. We are grateful to the conference sponsors for their support and to the USENIX staff for handling the conference logistics, marketing, and proceedings publication; it is a pleasure to work with them. Eddie Kohler and Geoff Voelker continue to provide invaluable service to the community by providing and supporting their HotCRP reviewing system and Banal format checker. Finally, we thank the NSDI '11 attendees and future readers of these papers: in the end, it is your interest in this work that makes all of these efforts worthwhile.

David G. Andersen, Carnegie Mellon University
Sylvia Ratnasamy, Intel Labs Berkeley

SSLShader: Cheap SSL Acceleration with Commodity Processors

Keon Jang+, Sangjin Han+, Seungyeop Han*, Sue Moon+, and KyoungSoo Park+

+*KAIST* **University of Washington*

Abstract

Secure end-to-end communication is becoming increasingly important as more private and sensitive data is transferred on the Internet. Unfortunately, today's SSL deployment is largely limited to security or privacy-critical domains. The low adoption rate is mainly attributed to the heavy cryptographic computation overhead on the server side, and the cost of good privacy on the Internet is tightly bound to expensive hardware SSL accelerators in practice.

In this paper we present high-performance SSL acceleration using commodity processors. First, we show that modern graphics processing units (GPUs) can be easily converted to general-purpose SSL accelerators. By exploiting the massive computing parallelism of GPUs, we accelerate SSL cryptographic operations beyond what state-of-the-art CPUs provide. Second, we build a transparent SSL proxy, SSLShader, that carefully leverages the trade-offs of recent hardware features such as AES-NI and NUMA and achieves both high throughput and low latency. In our evaluation, the GPU implementation of RSA shows a factor of 22.6 to 31.7 improvement over the fastest CPU implementation. SSLShader achieves 29K transactions per second for small files while it transfers large files at 13 Gbps on a commodity server machine. These numbers are comparable to high-end commercial SSL appliances at a fraction of their price.

1 Introduction

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) have served as de-facto standard protocols for secure transport layer communication for over 15 years. With endpoint authentication and content encryption, SSL delivers confidential data securely and prevents eavesdropping and tampering by random attackers. Online banking, e-commerce, and Web-based email sites typically employ SSL to protect sensitive user data such as passwords, credit card information, and private content. Operating atop the transport layer, SSL is used for various application protocols such as HTTP, SMTP, FTP, XMPP, and SIP, just to name a few.

Despite its great success, today's SSL deployment is largely limited to security-critical domains or enterprise applications. A recent survey shows that the total number of registered SSL certificates is slightly over one million [18], reflecting less than 0.5% of active Internet sites [19]. Even in the SSL-enabled sites, SSL is often enforced only for a fraction of activities (e.g., password submission or billing information). For example, Web-based email sites such as Windows Live Hotmail[1] and Yahoo! Mail[2] do not support SSL for the content, making the private data vulnerable for sniffing in untrusted wireless environments. Popular social networking sites such as Facebook[3] and Twitter[4] allow SSL only when users make explicit requests with a noticeable latency penalty. In fact, few sites listed in Alexa top 500 [2] enable SSL by default for the entire content.

The low SSL adoption is mainly attributed to its heavy computation overhead on the server side. The typical processing bottleneck lies in the key exchange phase involving public key cryptography [22, 29]. For instance, even the latest CPU core cannot handle more than 2K SSL transactions per second (TPS) with 1024-bit RSA while the same core can serve over 10K plaintext HTTP requests per second. As a workaround, high-performance SSL servers often distribute the load to a cluster of machines [52] or offload cryptographic operations to dedicated hardware proxies [3, 4, 6, 13] or accelerators [9, 10, 14, 15]. Consequently, user privacy in the Internet still remains an expensive option even with the modern processor innovation.

Our goal is to find a practical solution with commodity processors to bring the benefits of SSL to all private In-

[1] http://explore.live.com/windows-live-hotmail/

[2] http://mail.yahoo.com/

[3] http://www.facebook.com/

[4] http://www.twitter.com/

ternet communication. In this paper, we present our approach in two steps. First, we exploit commodity graphics processing units (GPUs) as high-performance cryptographic function accelerators. With hundreds of streaming processing cores, modern GPUs execute the code in the single-instruction-multiple-data (SIMD) fashion, providing ample computation cycles and high memory bandwidth to massively parallel applications. Through careful algorithm analysis and parallelization, we accelerate RSA, AES and SHA-1 cryptographic primitives with GPUs. Compared with previous GPU approaches that take hundreds of milliseconds to a few seconds to reach the peak RSA performance [37,56], our implementation produces the maximum throughput with one or two orders of magnitude smaller latency, which is well-suited for interactive Web environments.

Second, we build SSLShader, a GPU-accelerated SSL proxy that transparently handles SSL transactions for existing network servers. SSLShader selectively offloads cryptographic operations to GPUs to achieve high throughput and low latency depending on the load level. Moreover, SSLShader leverages the recent hardware features such as multi-core CPUs, the non-uniform memory access (NUMA) architecture, and the AES-NI instruction set.

Our contributions are summarized as follows:

(i) We provide detailed algorithm analysis and parallelization techniques to scale the performance of RSA, AES and SHA-1 in GPUs. To the best of our knowledge, our GPU implementation of RSA shows the highest throughput reported so far. On a single NVIDIA GTX580 card, our implementation shows 92K RSA operations/s for 1024-bit keys, a factor of 27 better performance over the fastest CPU implementation with a single 2.66 GHz Intel Xeon core.

(ii) We introduce opportunistic workload offloading between CPU and GPU to achieve both low latency and high throughput. When lightly loaded, SSLShader utilizes low-latency cryptographic code execution by CPUs, but at high load it batches and offloads multiple cryptographic operations to GPUs.

(iii) We build and evaluate a complete SSL proxy system that exploits GPUs as SSL accelerators. Unlike prior GPU work that focuses on microbenchmarks of cryptographic operations, we focus on systems interaction in handling the SSL protocol. SSLShader achieves 13 Gbps SSL large-file throughput handling 29K SSL TPS on a single machine with two hexa-core Intel Xeon 5650's.

The rest of the paper is organized as follows. In Section 2, we provide a brief background on SSL, popular cryptographic operations, and the modern GPU. In Sections 3 and 4 we explain our optimization techniques for RSA, AES and SHA-1 implementations in a GPU. In Sections 5 and 6, we show the design and evaluation of SSLShader. In Sections 7 and 8 we discuss related work and conclude.

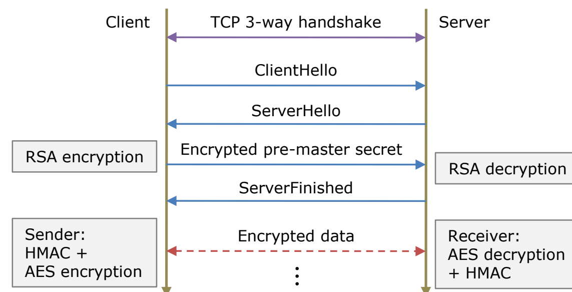


Figure 1: SSL handshake and data

2 Background

In this section, we provide a brief introduction to SSL and describe the cryptographic algorithms used in TLS_RSA_WITH_AES_128_CBC_SHA, one of the most popular SSL cipher suites. We also describe the basic architecture of modern GPUs and strategies to exploit them for cryptographic operations. In this paper we use TLS_RSA_WITH_AES_128_CBC_SHA as a reference cipher suite, but we believe our techniques to be easily applicable to other similar algorithms.

2.1 Secure Sockets Layer

SSL was developed by Netscape in 1994 and has been widely used for secure transport layer communication. SSL provides three important security properties in private communication: data confidentiality, data integrity, and end-point authentication. From SSL version 3.0, the official name has changed to TLS and the protocol has been standardized by IETF. SSL and TLS share the same protocol structure, but they are incompatible, since they use different key derivation functions to generate session and message authentication code (MAC) keys.

Figure 1 describes how the SSL protocol works. A client sends a ClientHello message to the target server with a list of supported cipher suites and a nonce. The server picks one (asymmetric cipher, symmetric cipher, MAC algorithm) tuple in the supported cipher suites, and responds with a ServerHello message with the chosen cipher suite, its own certificate and a server-side nonce. Upon receiving the ServerHello message, the client verifies the server's certificate, generates a pre-master secret and encrypts it with the server's public key. The encrypted pre-master secret is delivered to the server, and both parties independently generate two symmetric cipher session keys and two MAC keys using a predefined key derivation function with the pre-master key and the two nonces as input. Each (session, MAC) key pair is

used for encryption and MAC generation for one direction (e.g., client to server or server to client).

In the Web environment where most objects are small, the typical SSL bottleneck lies in decrypting the pre-master secret with the server-side private key. The client-side latency could increase significantly if the server is overloaded with many SSL connections. When the size of an object is large, the major computation overhead shifts to symmetric cipher execution and MAC calculation.

2.2 Cryptographic Operations

TLS_RSA_WITH_AES_128_CBC_SHA uses RSA, AES, and a Secure Hash Algorithm (SHA) based HMAC. Below we sketch out each cryptographic operation.

2.2.1 RSA

RSA [53] is an asymmetric cipher algorithm widely used for signing and encryption. To encrypt, a plaintext message is first transformed into an integer M, then turned into a ciphertext C with:

$$C := M^e \bmod n \tag{1}$$

with a public key (n, e). Decryption with a private key (n, d) can be done with

$$M := C^d \bmod n \tag{2}$$

C, M, d, and n are k-bit large integers, typically $1,024$, $2,048$, or even $4,096$ bits (or roughly 300, 600, or $1,200$ decimal digits). Since e is chosen to be a small number (common choices are 3, 17, and $65,537$), public key encryption is 20 to 60 times faster than private key decryption. RSA operations are compute-intensive, especially for SSL servers. Because servers perform expensive private key decryption for each SSL connection, handling many concurrent connections from clients is a challenge. In this paper we focus on private key RSA decryption, the main computation bottleneck on the server side.

2.2.2 AES

Advanced Encryption Standard (AES) [32] is a popular symmetric block cipher algorithm in SSL. AES divides plaintext message into 128-bit fixed blocks and encrypts each block into ciphertext with a 128, 192, or 256-bit key. The encryption algorithm consists of 10, 12, or 14 rounds of transformations depending on the key size. Each round uses a different round key generated from the original key using Rijndael's key schedule.

We implement AES encryption and decryption in cipher-block chaining (CBC) mode. In CBC mode, each plaintext block is XORed with a random block of the same size before encryption. The i-th block's random block is simply the $(i-1)$-th ciphertext block, and the initial random block, called the Initialization Vector (IV), is randomly generated and is sent in plaintext along with the encrypted message for decryption.

2.2.3 HMAC

Hash-based Message Authentication Code (HMAC) is used for message integrity and authentication. HMAC is defined as

$$HMAC(k,m) = H((k \oplus opad) \| H((k \oplus ipad) \| m)) \tag{3}$$

H is a hash function, k is a key, m is a message, and $ipad$ and $opad$ are predefined constants. Any hash function can be combined with HMAC and we use SHA-1 as it is the most popular.

2.3 GPU

Modern GPUs have hundreds of processing cores that can be used for general-purpose computing beyond graphics rendering. Both NVIDIA and AMD provide convenient programming libraries to use their GPUs for computation or memory-intensive applications. We use NVIDIA GPUs here, but our techniques are applicable to AMD GPUs as well.

A GPU executes code in the SIMD fashion that shares the same code path working on multiple data at the same time. For this reason, a GPU is ideal for parallel applications requiring high memory bandwidth to access different sets of data. The code that the GPU executes is called a *kernel*. To make full use of massive cores in a GPU, many threads are launched and run concurrently to execute the kernel code. This means more parallelism generally produces better utilization of GPU resources.

GPU kernel execution takes the following four steps: (*i*) the DMA controller transfers input data from host memory to GPU (device) memory; (*ii*) a host program instructs the GPU to launch the kernel; (*iii*) the GPU executes threads in parallel; and (*iv*) the DMA controller transfers the result data back to host memory from device memory.

The latest NVIDIA GPU is the GTX580, codenamed Fermi [20]. It has 512 cores consisting of 16 Streaming Multiprocessors (SMs), each of which has 32 Stream Processors (SPs or CUDA cores). In each SM, 48 KB shared memory (scratchpad RAM), 16 KB L1 cache, and $32,768$ 4-byte registers allow high-performance processing. To hide the hardware details, NVIDIA provides Compute Unified Device Architecture (CUDA) libraries to software programmers. CUDA libraries allow easy programming for general-purpose applications. More details about the architecture can be found in [47, 48].

The fundamental difference between CPUs and GPUs comes from how transistors are composed in the processor. A GPU devotes most of its die area to a large array of Arithmetic Logic Units (ALUs). In contrast, most CPU resources serve a large cache hierarchy and

a control plane for sophisticated acceleration of a single thread (e.g., out-of-order execution, speculative loads, and branch prediction), which are not much effective in cryptography. Our key insight of this work is that compute-intensive cryptographic operations can benefit from the abundant ALUs in a GPU, given enough parallelism (intra- and inter-flow).

3 Optimizing RSA for GPU

For RSA implementation on GPUs, the main challenge is to achieve high throughput while keeping the latency low. Naïve porting of CPU algorithms to a GPU would cause severe performance degradation, wasting most GPU computational resources. Since a single GPU thread runs at 10x to 100x slower speed than a CPU thread, the naïve approach would yield unacceptable latency.

In this section, we describe our approach and design choices to maximize performance of RSA decryption on GPUs. The key point in maximizing RSA performance lies in high parallelism. We exploit parallelism in the message level, in modular exponentiation, and finally in the word-size modular multiplication. We show that our parallel Multi-Precision (MP) algorithm obtains a significant gain in throughput and curbs latency increase to a reasonable level.

3.1 How to Parallelize RSA Operations?

Our main parallelization idea is to batch multiple RSA ciphertext messages and to split those messages into thousands of threads so that we can keep all GPU cores busy. Below we give a brief description of each level.

Independent Messages: At the coarsest level, we process multiple messages in parallel. Each message is inherently independent of other messages; no coordination between threads belonging to different messages is required.

Chinese Remainder Theorem (CRT): For each message, (2) can be broken into two independent modular exponentiations with CRT [51].

$$M_1 = C^{d \bmod (p-1)} \bmod p \tag{4a}$$

$$M_2 = C^{d \bmod (q-1)} \bmod q \tag{4b}$$

where p and q are $k/2$-bit prime numbers chosen in private key generation ($n = p \times q$). All four parameters, p, q, $d \bmod (p-1)$, and $d \bmod (q-1)$, are part of the RSA private key [38].

With CRT, we perform the two $k/2$-bit modular exponentiations in parallel. Each of which requires roughly 8 times less computation than k-bit modular exponentiation. Obtaining M from M_1 and M_2 adds only small overheads, compared to the gain from two $k/2$-bit modular exponentiations.

Large Integer Arithmetic: Since the word size of a computer is usually 32 or 64-bit, large integers must be broken into small multiple words. We can run multiple threads, each of which processes a word. However, we need carry-borrow processing or base extension in order to coordinate the outcome of per-word operations between threads. We consider two algorithms, standard Multi-Precision (MP) and Residue Number System (RNS), to represent and compute large integers. These algorithms are commonly used in software and hardware implementations of RSA.

3.2 Optimization Strategies

In our MP implementation we exploit the following two optimization strategies: (*i*) reducing the number of modular multiplications with the Constant Length Nonzero Windows (CLNW) partitioning algorithm; (*ii*) adopting Montgomery's reduction algorithm to improve the efficiency of each modular multiplication routine performed at each step of the exponentiation. These optimization techniques are also helpful for both serial software and hardware implementations, as well as for our GPU parallel implementations.

CLNW: With the binary square-and-multiply method, the expected number of modular multiplications is $3k/2$ for k-bit modular exponentiation [41]. For example, the expected number of operations for 512-bit modular exponentiation (used for 1024-bit RSA with CRT) is 768. The number can be reduced with sliding window techniques that scan multiple bits, instead of individual bits of the exponent.

We have implemented CLNW and reduced the number of modular multiplications from 768 to 607, achieving a 21% improvement [28]. One may instead use the Variable Length Nonzero Window (VLNW) algorithm [26], but it is known that VLNW does not give any performance advantage over CLNW on average [50].

Montgomery Reduction: In a modular multiplication $c = a \cdot b \bmod n$, an explicit k-bit modulo operation following a naïve multiplication should be avoided. Modulo operation requires a trial division by modulus n for the quotient, in order to compute the remainder. Division by a large divisor is very expensive in both software and hardware implementations and is not easily parallelizable, and thus inappropriate especially for GPUs.

Montgomery's algorithm allows a modular multiplication without a trial division [45]. Let

$$\overline{a} = a \cdot R \bmod n \tag{5}$$

be the *montgomeritized* form of a modulo n, where R and n are coprime and $n < R$. Montgomery multiplication

Algorithm 1 MMUL: Montgomery multiplication

Input: $\bar{a}, \bar{b}$
Output: $\bar{a} \cdot \bar{b} \cdot R^{-1} \bmod n$
Precomputation: R^{-1} such that $R \cdot R^{-1} \equiv 1 \pmod{n}$
n' such that $R \cdot R^{-1} - n \cdot n' = 1$
1: $T \leftarrow \bar{a} \cdot \bar{b}$
2: $M \leftarrow T \cdot n' \bmod R$
3: $U \leftarrow (T + M \cdot n)/R$
4: **if** $U \geq n$ **then**
5: **return** $U - n$
6: **else**
7: **return** U
8: **end if**

is defined as in Algorithm 1. If we set R to be 2^k, the division and modulo operations with R can be done very efficiently with bit shifting and masking.

Note that the result of Montgomery multiplication of $\overline{a}$ and $\overline{b}$ is still $\overline{a} \cdot \overline{b} \cdot R^{-1} \bmod n = \overline{a \cdot b} \bmod n$, the montgomeritized form of $a \cdot b$. For a modular exponentiation, we convert a ciphertext C into $\overline{C}$, get $\overline{C^d}$ with successive Montgomery multiplication operations, and invert it into $C^d \bmod n$. In this process, expensive divisions or modulo operations with n are eliminated.

The implementation of Montgomery multiplication depends on data structures used to represent large integers. Below we introduce our MP implementation.

3.3 MP implementation

The standard Multi-Precision algorithm is the most convenient way to represent large integers in a computer [41]. A k-bit integer A is broken into $s = \lceil k/w \rceil$ words of a_i's, where $i = 1, \ldots, s$ and w is typically set to the bit-length of a machine word (e.g., 32 or 64). Here we describe our MP implementation of Montgomery multiplication and various optimization techniques.

3.3.1 Multiplication

In Algorithm 1, the multiplication of two s-word integers appear three times in lines 1, 2, and 3. The time complexity of the serial multiplication algorithm that performs a shift-and-add of partial products is $O(s^2)$ (also known as the *schoolbook* multiplication). Implementation of an $O(s)$ parallel algorithm with linear scalability is not trivial due to carry processing. We have implemented an $O(s)$ parallel algorithm on s processors (threads) that works in two phases. In Figure 2, hiword and loword are high and low w bits of a $2w$-bit product respectively, and gray cells represent updated words by s threads. This parallelization scheme is commonly used for hardware implementation.

In the first phase, we accumulate $s \times 1$ partial products in $2s$ steps (s steps for each loword and hiword), ignoring any carries. Carries are accumulated in a separate array through the processing. Each step is simple enough to be translated into a small number of GPU instructions since it does not involve cascading carry propagation.

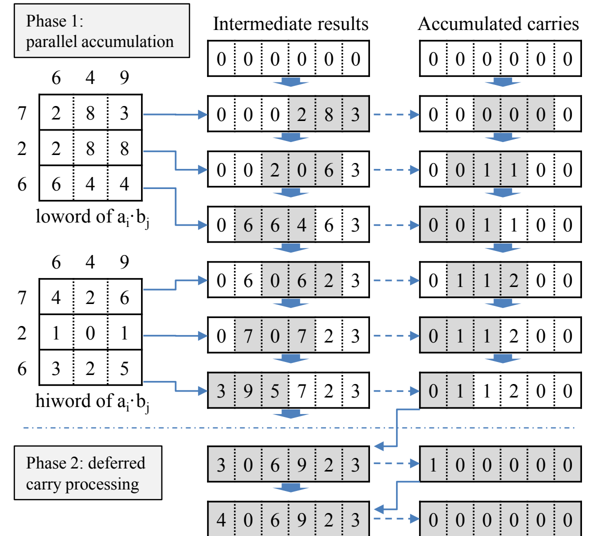


Figure 2: Parallel multiplication example of $649 \times 627 = 406,923$. For simplicity, a word holds a decimal digit rather than w-bit binary in the example.

The second phase repeatedly adds the carries to the intermediate result and renews the carries. This phase stops when all carries become 0, which can be checked in one instruction with the `__any()` voting function in CUDA [48]. The number of iterations is $s - 1$ in the worst case, but for most cases it takes one or two iterations since small carries (less than $2s$) rarely produce additional carries.

Our simple $O(s)$ algorithm is a significant improvement over the prior RSA implementation on GPUs. Harrison and Waldron parallelize $s \times s$ multiplications as follows [37]: Each of s threads independently performs $s \times 1$ multiplications in serial. Then s partial products are summed up in additive reduction in $\log n$ steps, each of which is done in serial as well. The resulting time complexity is $O(s \log s)$, and most of the threads are underutilized during the final reduction phase.

We also implemented RNS-based Montgomery multiplications. We adopt Kawamura's algorithm [40]. Even with extensive optimizations, the RNS implementation performs significantly slower than MP, and we use only the MP version in this paper. For future reference, we point out two main problems that we have encountered. First, CUDA does not support native integer division and modulo operations, on which the RNS Montgomery multiplication heavily depends. We have found that the performance of emulated operations is dependent on the size of a divisor and degrades significantly if the length of a divisor is longer than 14 bits. Second, since the number of threads is not a power of two, warps are not fully utilized and array index calculation becomes slow.

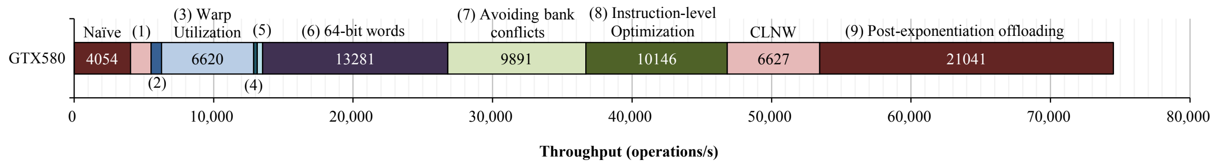


Figure 3: 1024-bit RSA performance with various optimization techniques. Sub-bars are placed in the same order as the techniques shown in Section 3.3.2, except for CLNW.

3.3.2 Optimizations

On top of CRT parallelization, CLNW, Montgomery reduction, modular exponentiation, and square-and-multiply optimization techniques, we conduct further optimizations as below. Figure 3 demonstrates how the overall throughput of the system increases as each optimization technique is applied. The *naïve* implementation includes CRT parallelization, basic implementation of Montgomery multiplication, and square-and-multiply modular exponentiation. For a 1024-bit ciphertext message with CRT, each of two 512-bit numbers (a ciphertext message) spans across 16 threads, each of which holds a 32-bit word, and those 16 threads are grouped as a CUDA block.

(1) Faster Calculation of $M \cdot n$: In Algorithm 1, the calculation of M and $M \cdot n$ requires two $s \times s$ multiplication operations. We reduce these into one $s \times 1$ and one $s \times s$ multiplication and interleave them in a single loop. This technique was originally introduced in [45], and we apply it for the parallel implementation.

(2) Interleaving $T + M \cdot n$: We interleave the calculation of $T + M \cdot n$ in a single multiplication loop. This optimization effectively reduces the overhead of loop construction and carry processing. This technique was used in the serial RSA implementation on a Digital Signal Processor (DSP) [34], and we parallelize it.

(3) Warp Utilization: In CUDA, a *warp* (a group of 32 threads in a CUDA block), is the basic unit of scheduling. Having only 16 threads in a block causes underutilization of warps, limiting the performance. We avoid this behavior by having blocks be responsible for multiple ciphertext messages, for full utilization of warps.

(4) Loop Unrolling: We unrolled the loop in Montgomery multiplication, by using the `#pragma unroll` feature supported in CUDA. Giving more optimization chances to the compiler is more beneficial than in CPU programming, due to the lack of out-of-order execution capability in GPU cores.

(5) Elimination of Divergency: Since threads in a warp execute the same instruction in lockstep, code-path divergency in a warp is expensive (all divergent paths must be taken in serial). For example, we minimize the divergency in our code by replacing `if` statements with flat arithmetic operations.

(6) Use of 64-bit Words: The native support for integer multiplication on GPUs, which is the basic building block of large integer arithmetic, has recently been added and is still evolving. GTX580 supports native single-cycle instructions that calculate hiword or loword of the product of two 32-bit integers.

Use of 64-bit words instead of 32-bit introduce a new trade-off on GPUs. While the multiplication of two 64-bit words takes four GPU cycles [48], it can halve the required number of threads and loop iterations depicted in Figure 2. We find that this optimization is highly effective when applied.

(7) Avoiding Bank Conflicts: The 64-bit access pattern to the intermediate results and carries in Figure 2 causes bank conflicts in shared memory between independent ciphertext messages in the same warp. We avoid this bank conflict by padding the arrays to adjust access pattern in shared memory.

(8) Instruction-Level Optimization: We have manually inspected and optimized the core code (about 10 lines) inside the multiplication loop, which is the most time-consuming part in our GPU code. We changed the code order at the CUDA C source level, until we got the desired assembly code. This includes the elimination of redundant instructions and pipeline stalls caused by Read-After-Write (RAW) register dependencies [47].

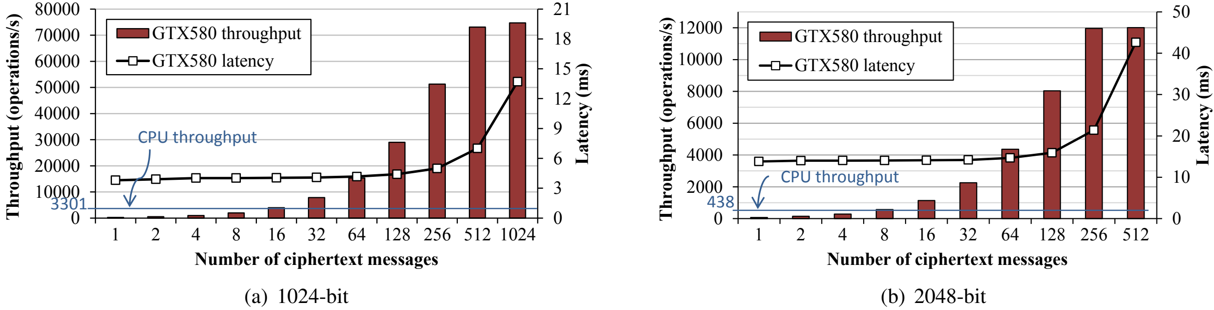
(9) Post-Exponentiation Offloading: Fusion of two partial modular exponentiation results from (4) is done on the CPU with the Mixed-Radix Conversion (MRC) algorithm as follows [27]:

$$M := M_2 + [(M_1 - M_2) \cdot (q^{-1} \bmod p)] \cdot q \qquad (6)$$

Although this processing is much lighter than modular exponentiation operations, the relative cost has become significant as we optimize the modular exponentiation process extensively. We have offloaded the above equation to the GPU, parallelizing at the message level. We also offload other miscellaneous processing in decryption such as integer-to-octet-string conversion and PKCS #1 depadding [38].

3.4 RSA Microbenchmarks

We compare our parallel RSA implementation to a serial CPU implementation. We use Intel Integrated Per-



(a) 1024-bit

(b) 2048-bit

Figure 4: RSA MP performance on a GTX580. A single core (Xeon X5650 2.66 GHz) is used for CPU performance.

	Processor	512	1024	2048	4096
Latency (ms)	CPU core	0.07	0.3	2.3	14.9
	GTX580, MP	1.1	3.8	13.83	52.46
Throughput (ops/s)	CPU core	13,924	3,301	438	67
	GTX580, MP	906	263	72	19
Peak (ops/s)	CPU core	13,924	3,301	438	67
	GTX580, MP	322,167	74,732	12,044	1,661

Table 1: RSA performance with various key sizes

formance Primitives (IPP) [8] as a CPU counterpart. IPP is the fastest implementation we have tried, outperforming other publicly available libraries for all key sizes. It performs 3,301 RSA decryption operations/s for a 1024-bit key on a 2.66 GHz CPU core. Since this number is higher than what Kounavis *et al.* recently report (2,990 operations/s on a 3.00 GHz CPU core) in [43], we believe its CPU reference implementation is a fair comparison to our GPU code.

Table 1 summarizes the performance of RSA on the CPU (a single 2.66 GHz core) and GTX580. With only one ciphertext message per launch, the GPU's performance shows an order of magnitude worse throughput (operations per second) and latency (the execution time). Given enough parallelism, however, the GPU produces much higher throughput than the CPU. The MP implementation on the GTX580 shows 23.1x, 22.6x, 27.5x, and 31.7x speedup compared with a single CPU core, for 512-bit, 1024-bit, 2048-bit, and 4096-bit RSA, respectively. The performance gains are comparable to what we expect from three hexa-core CPUs.

Figure 4 shows the correlation between latency and throughput of our MP implementation. The throughput improves as the number of concurrent messages grows, but reaches a plateau beyond 512 messages. The latency increases very slowly, but grows linearly with the number of messages beyond the point where the GPU is fully utilized. Even at peak throughput the latency stays below 7 to 13.7 ms for more than 70,000 operations/s for 1024-bit RSA decryption on a GTX580 card.

Many cipher algorithms, such as DSA [5], Diffie Hellman key exchange [33], and Elliptic Curve Cryptography (ECC) [42], depend on modular exponentiation as well as RSA. Our optimization techniques presented in Section 3 are applicable to those algorithms and can offer an efficient platform for their GPU implementation.

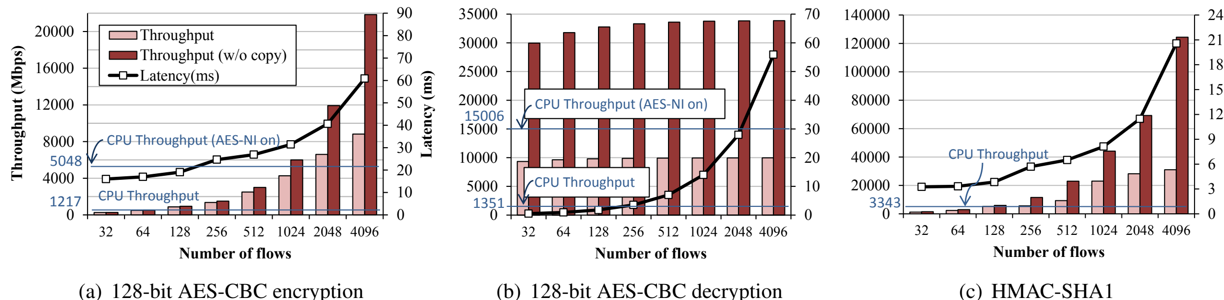
We summarize our RSA implementation on GPUs. First, the parallel RSA implementation on a GPU brings significant throughput improvement over a CPU. Second, we need many ciphertext messages in a batch for full utilization of GPUs with enough parallelism, in order to take a performance advantage over CPUs. In Section 5.4 we introduce the concept of *asynchronous concurrent execution*, which allows smaller batch sizes and thus shorter queueing and processing latency, while yielding even better throughput. Lastly, while the GPU implementation shows reasonable latency, it still imposes perceptible delay for SSL clients. This problem is addressed in Section 5.2 with *opportunistic offloading*, which exploits the CPU for low latency when under-loaded and offloads to the GPU for high throughput when a sufficient number of operations are pending.

4 Accelerating AES and HMAC-SHA1

4.1 GPU-accelerated AES

Since CBC mode encryption has a dependency on the previous block result, AES encryption in the same flow is serialized. On the other hand, decryption can be run concurrently as the previous block result is already known at decryption time. Therefore, AES-CBC decryption in a GPU runs much faster than AES-CBC encryption.

We have implemented AES for a GPU with the following optimizations. As shown in [36], on-chip shared memory offers two orders of magnitude faster access time than global memory on GPU. To exploit this fact, at the beginning of the AES cipher function, each thread copies part of the lookup table into shared memory. Additionally, we have chosen to derive the round key at each round, instead of using pre-expanded keys from global memory. Though it incurs more computation overhead, it avoids expensive global memory access and reduces the total latency.



(a) 128-bit AES-CBC encryption (b) 128-bit AES-CBC decryption (c) HMAC-SHA1

Figure 5: AES and HMAC-SHA1 performance on GTX580. A single core (Xeon X5650 2.66 GHz) is used for CPU performance.

4.2 AES-NI

Intel has recently added the AES instruction set (AES-NI) to the latest lineup of x86 processors. AES-NI runs one round of AES encryption or decryption with a single instruction (AESENC or AESDEC), and its performances for AES-GCM and AES-CTR are 2.5 to 6 times faster than a software implementation [7, 39]. AES-NI is especially useful for handling large files since data transfer overhead between host and device memory quickly becomes the bottleneck for GPU-accelerated AES. However, GPU-based symmetric cipher offloading still provides a benefit, if (i) CPUs do not support AES-NI, (ii) CPUs become the bottleneck handling the network stack and other server code, or (iii) other cipher functions (such as RC4 and 3DES) are needed.

4.3 GPU-accelerated HMAC-SHA1

The performance of HMAC-SHA1 depends on SHA1. Thus, we focus on the SHA1 algorithm. SHA1 takes 512 bits at each round and generates a 20-byte digest. Each round uses the previous round's result; thus SHA1 can not be run in parallel within a single message. Our SHA1 optimization in a GPU is divided into two parts: (i) reducing memory access by processing data in the register, and (ii) reducing required memory footprint to fit in the GPU registers.

Each round of SHA-1 is divided into four different steps, and at each step it processes 20 32-bit words; in total, 80 intermediate 32-bit values are used. A typical CPU implementation pre-calculates all 80 words before processing, which requires a 320-byte buffer. However, the algorithm only depends on the previous 16 words at any time. We calculate each intermediate data on demand, thus reducing the memory requirement to 64 bytes, which fits into the registers.[5]

To avoid global memory allocation, we unroll all loops and hardcode the buffer access with constant indices. This way the compiler register-allocates all the necessary 16 words. With this approach we see about 100% performance improvement over the naïve implementation.

4.4 Microbenchmarks

Figure 5 compares the throughput and latency results of AES and HMAC-SHA1 with one GTX580 card and one 2.66 GHz CPU core. For the CPU implementations, we use Intel IPP, which shows the best performance of AES and SHA-1 as of writing this paper. We fix the flow length to 16 KB, the largest record size in SSL, and vary the number of flows from 32 to 4,096. Our AES-CBC implementation shows the peak performance of 8.8 Gbps and 10.0 Gbps for encryption and decryption respectively when we consider the data transfer cost, but the numbers go up to 21.9 Gbps and 33.9 Gbps without the copy cost. AES-NI shows 5 Gbps and 15 Gbps even with a single CPU core and thus one or two cores would exceed our GPU performance. Our GPU version matches 6.5 and 7.4 CPU cores without AES-NI support for encryption and decryption. For HMAC-SHA1, our GPU implementation shows 31 Gbps with the data transfer cost and 124 Gbps without, and matches the performance of 9.4 CPU cores.

Our findings are summarized as follows. (i) AES-NI shows the best performance per dollar, (ii) the data transfer cost in GPU reduces the performance by a factor of 3.39 and 4 in AES and HMAC-SHA1, and (iii) the GPU helps in offloading HMAC-SHA1 and AES workloads when CPUs do not support AES-NI. Since a recent hardware trend shows that the GPU cores are being integrated into the CPU [16, 54], we expect the impact of the data transfer overhead will decrease in the near future.

5 SSLShader

We build a scalable SSL reverse proxy, SSLShader, to incorporate the high-performance cryptographic operations using a GPU into SSL processing. Though proxying generally incurs redundant I/O and data copy overheads, we choose transparent proxing because it provides the SSL acceleration benefit to existing TCP-based

[5] The idea to reduce the memory footprint is from a Web post in an NVIDIA forum: http://forums.nvidia.com/index.php?showtopic=102349

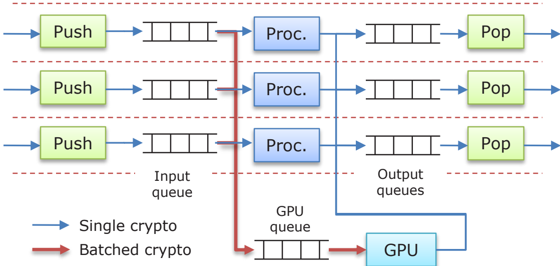


Figure 6: Overview of SSLShader

servers without code modification. SSLShader interacts directly with the SSL clients while communicating with the back-end server in plaintext. We assume that the SSLShader-to-server communication takes place in a secure environment, but one can encrypt the back-end traffic with a shared symmetric key in other cases.

The design goal of SSLShader is twofold. First, the performance should scale well to the number of CPU and GPU cores. Second, SSLShader should curb the latency to support interactive environments while improving the throughput at high load. In this section we outline the key design features of SSLShader.

5.1 Basic Design

Figure 6 depicts the overall architecture of SSLShader. SSLShader is implemented in event-driven threads. To scale with multi-core CPUs, it spawns one worker thread per CPU core and each thread accepts and processes client-side SSL connections independently. Each connection is accepted and processed by the same thread to avoid cache bouncing between CPU cores. SSLShader also creates one GPU-interfacing thread per GPU that launches GPU kernel functions to offload cryptographic operations.

Each cryptographic operation type (RSA, AES, HMAC-SHA1) has its own request input queue per worker thread. Cryptographic operations of the same type are inserted into the same queue, and are moved to a queue in the GPU-interfacing thread when the input queue size exceeds a certain threshold value. GPU-interfacing threads simply offload the requested operations in a batch by launching GPU kernels. The results are placed back on a per-worker thread output queue so that the worker thread can resume the execution of the next step in SSL processing.

5.2 Opportunistic Offloading

In order to fully exploit the parallelism, we should batch multiple cryptographic operations and offload them to the GPU. On the GTX580, the peak 1024-bit RSA performance is achieved when batching 256-512 operations, that is, handling 256-512 concurrent SSL connections.

Cryptographic operation	Minimum	Maximum
RSA (1024-bit)	16	512
AES128-CBC Decryption	32 (2,048)	2,048
AES128-CBC Encryption	128 (2,048)	2,048
HMAC-SHA1	128	2,048

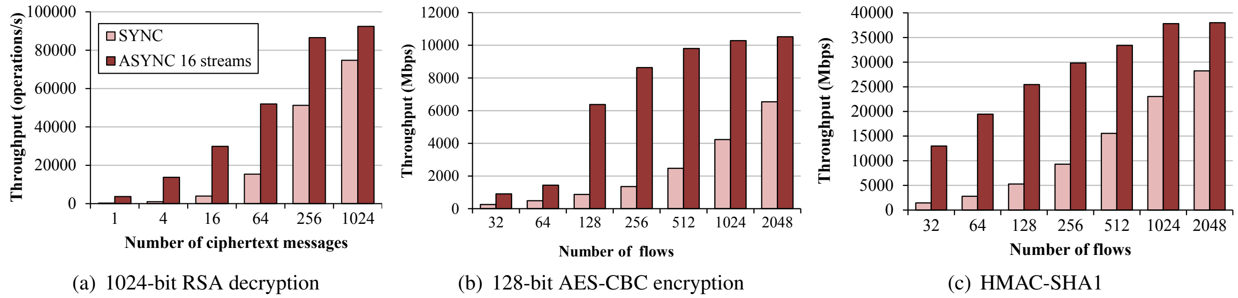
Table 2: Thresholds for GPU cryptographic operations per single kernel launch. Numbers in parenthesis are thresholds when AES-NI is used.

While batching generally improves the GPU throughput, a naïve approach of batching a fixed number of operations would increase processing latency when the load level is low.

We propose a simple GPU offloading algorithm that reduces response latency when lightly loaded and improves the overall throughput at high load. When a worker thread inserts a cryptographic request to an input queue, it first checks the number of the same type of requests in all workers' queues, and its minimum batching threshold (the number of queued requests required for GPU offloading). If the number of requests is above the threshold, SSLShader moves all the requests in the worker thread queue to a GPU-interfacing thread queue. The batching thresholds are determined based on the GPU's throughput. The minimum threshold is set when the GPU performs better than a single CPU core, and the maximum is set when the maximum throughput is achieved. We limit maximum batch size since pushing too many requests into a queue in the GPU-interfacing thread could result in long delay without throughput improvement. The thresholds can be drawn automatically from benchmark tests at configuration time. For AES, thresholds are different when AES-NI is enabled. If AES-NI is available, we set the minimum threshold to be the same as the maximum threshold, hoping to benefit from extra processing power only when the CPU is overloaded. Table 2 shows the thresholds we use with the GTX580.

For low latency and high parallelism, the worker thread prioritizes I/O events, and processes cryptographic operations when it has no I/O event. Worker threads handle cryptographic operations in the first-in first-out (FIFO) manner. We put a timestamp on each cryptographic request as it arrives, and use the timestamp to find the earliest arrived operation. The GPU also uses FIFO scheduling for processing cryptographic operations. The GPU-interfacing thread looks at the head timestamp of requests by the type, and processes the earliest request's type in a batch. Sometimes it takes too long for the worker thread to drain the cryptographic operations in its queue and this can lead to I/O starvation. To prevent this, we have worker threads periodically check for I/O events while processing cryptographic operations.

We also tested priority-based scheduling by having the



(a) 1024-bit RSA decryption (b) 128-bit AES-CBC encryption (c) HMAC-SHA1

Figure 7: Performance improvement from asynchronous concurrent execution with 16 streams, independent CUDA contexts of commands that execute in order asynchronously. Each flow size is 16KB for (b) and (c).

CPU prioritize HMAC-SHA1 and AES encryption, and the GPU prioritize RSA and AES decryption. This strategy often improves the peak throughput, but we reject this idea because lower-priority cryptographic operations could suffer from starvation, and we noticed unstable throughput and longer latency in many cases.

5.3 NUMA-aware GPU Sharing

NUMA systems are becoming commonplace in server machines. In NUMA systems, the communication cost between CPU cores varies greatly, depending on the number of NUMA hops. For high scalability, it is necessary to reduce inter-core communication by careful NUMA-aware data placement.

When we use a GPU, we should consider the following issues: (i) GPUs are known to perform badly when used by multiple threads simultaneously due to high context switching overhead [48]; (ii) gathering cryptographic operations from multiple cores brings more parallelism and helps to exploit the full GPU capacity; and (iii) memory access or synchronization across NUMA nodes is much more expensive than intra-NUMA node operation. For these reasons, we limit the sharing of GPUs to the threads in the same NUMA node.

For intra-NUMA node communication we choose threads over processes for faster sharing of the queues as offloading cryptographic operations requires data movement between worker and GPU-interfacing threads. For inter-NUMA node communication, we choose processes for ease of connection handling without kernel lock contention at socket creation and closure.

5.4 Asynchronous Concurrent Execution

The most recent CUDA device with Compute Capability 2.0 provides concurrent GPU kernel execution and data copy for better utilization of the GPU. On the GTX580, up to sixteen different kernels can run simultaneously within a single GPU, and copies from device to host and host to device can overlap with each other as well as with kernel execution To benefit from concurrent execution and copy, SSLShader launches all GPU transactions asynchronously. With asynchronous concurrent execution, we see up to $1,399\%$, 731%, and 890% performance improvements over synchronous execution in RSA, AES encryption and HMAC-SHA1, respectively. Figure 7 depicts the effect of asynchronous concurrent execution by varying the batch size. When the batch size is small, asynchronous concurrent execution improves performance greatly as idle GPU cores can be better utilized. But even for a large batch size such as $2,048$, we see $30 \sim 60\%$ performance improvement in HMAC-SHA1 and AES. The overlap of DMA data copy and kernel execution improves the performance even when all cores in the GPU are already utilized. In RSA, the performance improvement in the batch size of 1024 is fairly small compared to those of AES or HMAC-SHA1 because the data copy time in RSA is relatively smaller than the execution time and the GPU is sufficiently utilized with large batch sizes.

We believe our design and implementation strategies in this section are not limited to only SSLShader, and can be applied to any applications that want to exploit the massive parallelism of GPUs. While none of the techniques are ground-breaking, their combination brings a drastic difference in the utilization of GPUs, latency reduction, and throughput improvement.

6 Evaluation

In this section we evaluate the effectiveness of SSLShader using HTTPS, the most popular protocol that uses SSL. We show that SSLShader achieves high performance in connection handling and large-file data transfer with small latency overhead.

6.1 System Configuration

Our server platform is equipped with two Intel Xeon X5650 (hexa-core 2.66 GHz) CPUs, 24 GB memory, and two NVIDIA GTX580 cards (512 cores, 1.5 GHz,

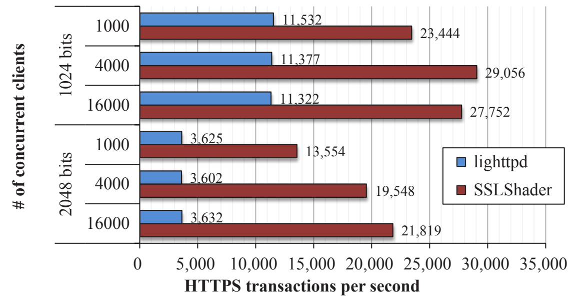


Figure 8: Transactions per second

Image Name	CPU Usage (%)
Kernel NIC device driver	2.32
Kernel (including TCP/IP stack)	60.35
SSLShader	5.31
libc (memory copy and others)	9.88
IPP + libcrypto (cryptographic operations)	12.89
`lighttpd` (back-end web server)	4.90
others	4.35

Table 3: CPU usage breakdown using oprofile

1.5 GB RAM). We install Ubuntu Linux 10.04, NVIDIA CUDA Driver v256.40, and Intel `ixgbe`[6] driver v2.1.4 on the server. As back-end web server software, we run `lighttpd`[7] v1.4.28 with 12 worker processes to match the number of CPU cores. In all experiments, we run `lighttpd` and SSLShader on the same machine.

We compare SSLShader against lighttpd with OpenSSL. For fair comparison, we spent a fair amount of time to patch OpenSSL 1.0.0 to use IPP 7.0 which has AES-NI support as well as the latest RSA and SHA-1 optimizations. We find that IPP 7.0 improves the RSA, AES, and HMAC performance by 55%, 10%, and 22% respectively from the OpenSSL 1.0.0 default implementation. As our goal is to offload SSL computation overhead, we focus on static content to prevent the back-end web server from becoming a bottleneck. To generate HTTP requests, we run the Apache HTTP server benchmark tool (`ab`) [1] on seven 2.66 GHz Intel Nehalem quad-core client machines. We modified `ab` to support rate-limiting and to report latency for each connection.

6.2 SSL Handshake Performance

To evaluate the performance of connection handshake, we measure the number of SSL transactions per second (TPS) for a small HTTP object (43 bytes including HTTP headers). Figure 8 shows the maximum TPS achieved by varying the number of concurrent connections. For 1024-bit RSA keys, SSLShader achieves 29K TPS, which is 2.5 times faster than 11.2K TPS for `lighttpd` with OpenSSL. SSLShader achieves 21.8K TPS, for 2048-bit RSA, which is 6 times higher than 3.6K TPS of `lighttpd`. Given that 768-bit RSA was cracked early in 2010 [12] and that NIST recommends 2048-bit RSA for secure operations as of 2011 [46], the large performance improvement with 2048-bit RSA is significant. In SSLShader, the throughput increases as the concurrency increases because the GPU can exploit more parallelism. In 2048-bit RSA, 21.8K is close to the peak throughput of 24.1K msg/s with two GTX580s, meaning that the GPUs are almost fully utilized. However, the performance of RSA 1024-bit is much less than the peak throughput of a single GPU, which implies that the GPUs have idle computing capacity.

We run `oprofile` to analyze the bottleneck for the RSA 1024-bit case with 16,000 concurrent clients. Table 3 summarizes where the CPU cycles are spent. We see that more than 60% of CPU time is spent in the kernel for accepting connections and networking I/O; 13% of the CPU cycles are spent for cryptographic operations, mostly for the Pseudo Random Function (PRF) used for generating session keys from the master secret in the handshake step. We chose not to offload PRF to GPUs because it is run only once in the handshake step and its computation overhead is less than 1/10th of the RSA decryption overhead. We conclude that the performance bottleneck is in the Linux kernel that does not scale to multi-core CPUs for connection acceptance, as is also observed in [57].

6.3 Response Time Distribution

Naïvely using a GPU for cryptographic operations could lead to high latency when the load level is low. Opportunistic offloading guards against this problem, minimizing the latency when the load is light and maximizing the throughput when the load is high. To evaluate the effectiveness of our opportunistic offloading algorithm, we measure the response time for both heavy and light load cases. We control the load by rate-limiting the clients. For `lighttpd`, we set the limits to 1K TPS for light load and 11K TPS for heavy load. For SSLShader, we further increase the heavy load limit to 29K TPS. For heavy load experiments, we vary the maximum number of clients from 1K to 4K. Clients repeatedly request the small HTTP objects as in the handshake experiment.

Figure 9 shows the cumulative distribution functions (CDFs) of response times. When the load level is low, both `lighttpd` and SSLShader handle most of the connections in a few milliseconds (ms), which shows that the opportunistic offloading algorithm intentionally uses the CPU to benefit from its low latency. SSLShader shows a slight increase in response time (2 ms vs. 3 ms on median) due to the proxying overhead. At heavy load

[6] http://sourceforge.net/projects/e1000/files/ixgbe%20stable/
[7] http://www.lighttpd.net/

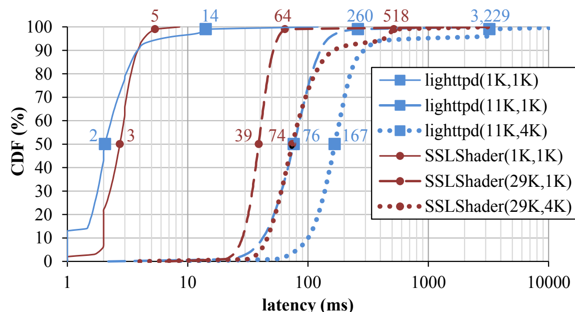


Figure 9: Latency distribution in the overloaded case. Numbers in parenthesis represent the maximum requests rate and the maximum concurrency.

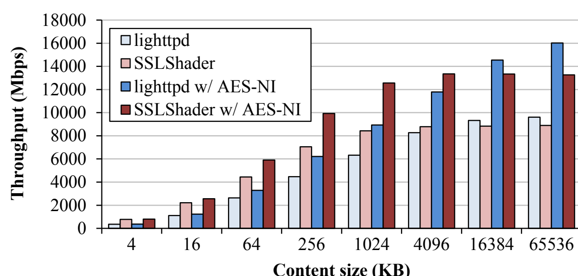


Figure 10: Bulk transfer throughput

with 1K concurrent connections, SSLShader's latency is lower than that of lighttpd because CPUs are overloaded and lighttpd produces longer response times. In contrast, SSLShader reduces the CPU overhead by offloading the majority of cryptographic operations to GPUs. SSLShader shows 39 ms and 64 ms for 50th and 99th percentiles while lighttpd shows 76 ms and 260 ms each even at the much lower TPS load level. Even if we increase the load with 4K concurrent clients, 70% of SSLShader response times remain similar to those of lighttpd with 1K clients at the 11K TPS level.

6.4 Bulk Data Encryption Performance

We measure bulk data encryption performance by varying the file size from 4 KB to 64 MB with and without AES-NI support, and show the results in Figure 10. When AES-NI is enabled, the SSLShader throughput peaks at 13 Gbps while lighttpd peaks at 16.0 Gbps. We note that increasing the content size above 64 MB does not increase lighttpd's throughput. For contents smaller than 4 MB, SSLShader performs 1.3 to 2.2x better than lighttpd while lighttpd shows 1.1x to 1.2x better performance for contents larger than 4 MB. As the content size grows and throughput increases, the proxying overhead increases accordingly, and eventually becomes the performance bottleneck. With oprofile, we find that 30% of CPU time is spent on data copying, and 20% is spent on handling interrupts for SSLShader, leaving only 50% for use in cryptographic operation and other processing. Without AES-NI, SSLShader achieves 8.9 Gbps, while lighttpd achieves 9.6 Gbps. The peak throughput of SSLShader is slightly lower due to the copying overhead as well.

Considering typical Web objects and email contents are smaller than 100 KB [21, 23], we believe that the performance gain in small content size and the benefit of transparent proxying outweigh the small performance loss in large files in many real-world scenarios. Also, the GPU is starting to be integrated into the CPU as in AMD's Fusion [16], and we expect that such technology will mitigate the performance problem by eliminating the data transfer between GPU and CPU.

7 Discussion & Related Work

SSL Performance: SSL performance analysis and acceleration have drawn much attention in the context of secure Web server performance. Earlier, Apostolopoulos *et al.* analyzed the SSL performance of Web servers using the SpecWeb96 benchmark tool [22]. They observe that the SSL-enabled Web servers are up to two orders of magnitude slower than plaintext Web servers. For small HTTP transactions, the main bottleneck lies in SSL handshaking while data encryption takes up significant CPU cycles when the content gets larger. Later, Coarfa *et al.* reported similar results and estimated the upper bound in the performance improvement with each SSL operation optimization [29]. To improve the SSL handshake performance, Boneh *et al.* proposed client-side caching of server certificates to reduce the SSL round-trip overhead [25]. Another approach is to process multiple RSA decryptions in a batch using Fiat's batch RSA [55]. They report a 2.5x speedup on their Web server experiments by batching four RSA operations.

Recently, Kounavis *et al.* improve the SSL performance with general-purpose CPUs [43]. They optimize the schoolbook big number multiplication and benefit from AES-NI for symmetric cipher. To reduce the CPU overhead for MAC algorithms, they use the Galois Counter Mode (GCM) which combines the AES encryption with the authentication. In comparison, we argue that GPUs bring extra computing power in a cost-effective manner, especially for RSA and HMAC-SHA1. By parallelizing the schoolbook multiplication and various optimizations, our 1024-bit RSA implementation on a GPU shows 30x improvement over their 3.0 GHz CPU core. Further, we focus on TLS 1.0 which is widely used in practice, whereas GCM is only supported in TLS 1.2, which is not popular yet.

AES Implementations on GPU: Modern GPUs are attractive for computation-intensive AES operations [30, 31, 36, 44, 49, 58]. Most GPU-based implementations ex-

ploit shared memory and on-demand round key calculation to reduce the global memory access. However, we find few references that evaluate the AES performance in the CBC mode. Unlike electronic codebook (ECB) mode or counter (CTR) mode, the CBC mode is hard to parallelize but is most widely used. Also, most of them report the numbers without data copy overhead, but we find the copy overhead severely impairs the AES performance.

Manavski *et al.* report 8.28 Gbps AES performance on GTX 280 (240 cores, 1.296 GHz) [44], while Osvik *et al.* report 30.9 Gbps on half of a GTX 295 (2 x 240 cores, 1.242 GHz) [49]. Both of them use the ECB mode without data copy overhead. In the same setting, our implementation shows 32.8 Gbps on GTX 285 (240 cores, 1.476 GHz). Direct comparison is hard due to different GPUs, but our number is comparable to these results (3.48x that of Manavski's, 0.89x that of Osvik's) by the cycles per byte metric.

RSA Implementations on GPU: Szerwinski and Güneysu made the first implementation of RSA on the general-purpose GPU computation framework [56]. They reported two orders of magnitude lower performance than ours, but it should not be directly compared because they used a relatively old NVIDIA 8800GTS card with a different GPU architecture.

Harrison and Waldron report on 1024-bit key RSA implementation on an NVIDIA GPU [37], and to the best of our knowledge theirs is the fastest implementation before our work. They compare serial and MP parallel approaches in Montgomery multiplication and conclude that the parallel implementation shows worse performance at scale due to GPU thread synchronization overhead. We have run their serial code on an NVIDIA GTX580 card, and found that their peak throughput reaches $31,220$ operations/s at a latency of 131 ms with $4,096$ messages per batch. Our throughput on the same card shows $74,733$ operations/s at a latency of 13.7 ms with 512 messages per batch, 2.3x improvement in throughput and 9.6x latency reduction.

Comparison with H/W Acceleration Cards: Many SSL cards support OpenSSL engine API [11] so that their hardware can easily accelerate existing software. Current hardware accelerators support 7K to 200K RSA operations/s with 1024-bit keys [10, 14]. Our GPU implementation is comparable with these high-end hardware accelerators, running at up to 92K RSA operations/s at much lower cost. Moreover, GPUs are flexible for adoption of new cryptographic algorithms.

Other Protocols for Secure Communication: Bittau *et al.* propose *tcpcrypt* as an extension of TCP for secure communication [24]. *Tcpcrypt* is essentially a clean-slate redesign of SSL that shifts the decryption overhead by private key to clients and that allows a range of authentication mechanisms. Their evaluation reports 25x better connection handling performance when compared with SSL. Moreover, *tcpcrypt* provides forward secrecy by default while SSL leaves that as an option. While fixing the SSL protocol is desirable, we focus on improving the current practice of SSL in this work. IPsec [17] provides secure communication at the IP layer, which is widely used for VPN. IPsec can be more easily parallelized compared to SSL since many packets can be processed in parallel [35].

Performance per $ Comparison: In Table 4, we show the price and relative performance to price for two CPUs, GTX580, and a popular SSL accelerator card. Intel Xeon X5650 and GTX580 are choices for our experiments. i7 920 has four CPU cores with the same clock speed as the X5650 without AES-NI support. We choose the CN1620[8] because it is one of the most cost-effective accelerators that we have found. Though it is difficult to compare the performance per dollar directly (e.g., GPU cannot be used without CPU), we present the information here to get the sense of cost effectiveness for each hardware.

	Price ($)	RSA (ops/sec/$)	AES-ENC (Mbps/$)	AES-DEC (Mbps/$)	SHA1 (Mbps/$)
X5650	996	19.9	30.6	92.2	20.2
i7 920	288	45.8	18.9	19.0	46.5
GTX580	499	185.3	21.3	25.1	62.3
CN1620	2,129	30.5	2.8	2.8	2.8

Table 4: Performance per $ comparison (price as of Feb. 2011)

GTX580 shows the best performance per dollar for RSA and HMAC-SHA1. For AES operations, X5650 is the best with its AES-NI capability, and GTX580 shows a slightly better number compared to i7 920. CN1620 is inefficient in terms of performance per dollar for all operations. SSL accelerators typically have much better power efficiency compared to general purpose processors and it is mainly used in high-end network equipment rather than on server machines.

8 Conclusions

We have enjoyed the security of SSL for over a decade and it is high time that we used it for every private Internet communication. As a cheap way to scale the performance of SSL, we propose using graphics cards as high-performance SSL accelerators. We have presented a number of novel techniques to accelerate the cryptographic operations on GPUs. On top of that, we have built SSLShader, an SSL reverse proxy, that opportunistically offloads cryptographic operations to GPUs and achieves high throughput and low response latency.

[8] Model name is CN1620-400-NHB4-E-G and more details are on `http://www.scantec-shop.com/cn1620-400-nhb4-e-g-375.html`

Our evaluation shows that we can scale 1024-bit RSA up to 92K operations/s with a single GPU card by careful workload pipelining. SSLShader handles 29K SSL TPS and achieves 13 Gbps bulk encryption throughput on commodity hardware. We hope our work pushes SSL to a wider adoption than today.

We report that inefficiency in the Linux TCP/IP stack is keeping performance lower than what SSLShader can potentially offer. Most of the inefficiency in the Linux TCP/IP stack comes from mangled flow affinity and serialization problems in multi-core systems. We leave these issues to future work.

9 Acknowledgment

We thank Geoff Voelker, anonymous reviewers, and our shepherd David Mazières for their help and invaluable comments. This research was funded by NAP of Korea Research Council of Fundamental Science & Technology, MKE (Ministry of Knowledge Economy of Repbulic of Korea, project no. 10035231-2010-01), KAIST ICC, and KAIST High Risk High Return Project (HRHRP).

References

[1] ab - Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/docs/2.2/en/programs/ab.html.

[2] Alexa Top 500 Global Sites. http://www.alexa.com/topsites.

[3] Application Delivery Controllers, Array Networks. http://www.arraynetworks.net/?pageid=365.

[4] Content Services Switches, Cisco. http://www.cisco.com/web/go/css11500.

[5] Digital Signature Standard. http://csrc.nist.gov/fips.

[6] F5 BIG-IP SSL Accelerator. http://www.f5.com/products/big-ip/feature-modules/ssl-acceleration.html.

[7] Intel Advanced Encryption Standard Instructions (AES-NI). http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/.

[8] Intel Integrated Performance Primitives. http://software.intel.com/en-us/intel-ipp/.

[9] nFast Series, Thales. http://iss.thalesgroup.com/Products/.

[10] NITROX security processor, Cavium Networks. http://www.caviumnetworks.com/processor_security_nitrox-III.html.

[11] OpenSSL Engine. http://www.openssl.org/docs/crypto/engine.html.

[12] Researchers crack 768-bit RSA. http://www.bit-tech.net/news/bits/2010/01/13/researchers-crack-768-bit-rsa/1.

[13] ServerIron ADX Series, Brocade. http://www.brocade.com/products-solutions/products/application-delivery/serveriron-adx-series/index.page.

[14] Silicom Protocol Processor Adapter. http://www.silicom-usa.com/default.asp?contentID=676.

[15] SSL Acceleration Cards, CAI Networks. http://cainetworks.com/products/ssl/rsa7000.htm.

[16] The AMD Fusion Family of APUs. http://sites.amd.com/us/fusion/APU/Pages/fusion.aspx.

[17] Security Architecture for the Internet Protocol. RFC 4301, 2005.

[18] Netcraft SSL Survey. http://news.netcraft.com/SSL-survey, 2009.

[19] Netcraft Web Server Survey. http://news.netcraft.com/archives/2010/04/15/april_2010_web_server_survey.html, 2009.

[20] NVIDIA's Next Generation CUDA™Compute Architecture: Fermi™. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009.

[21] S. Agarwal, V. N. Padmanabhan, and D. A. Joseph. Addressing email loss with suremail: Measurement, design, and evaluation. In *USENIX ATC*, 2007.

[22] G. Apostolopoulos, V. Peris, and D. Saha. Transport Layer Security: How much does it really cost? In *IEEE Infocom*, 1999.

[23] A. Badam, K. Park, V. Pai, and L. Peterson. Hashcache: Cache storage for the next billion. In *NSDI*, 2009.

[24] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. The case for ubiquitous transport-level encryption. In *USENIX Security Symposium*, 2010.

[25] D. Boneh, H. Shacham, and E. Rescrola. Client side caching for TLS. In *Network and Distributed System Security Symposium (NDSS)*, 2002.

[26] J. Bos and M. Coster. Addition chain heuristics. In *Advances in Cryptology (CRYPTO)*, 1989.

[27] Ç. K. Koç. High-speed RSA implementation. Technical Report, 1994.

[28] Ç. K. Koç. Analysis of sliding window techniques for exponentiation. *Computer and Mathematics with Applications*, 30(10):17–24, 1995.

[29] C. Coarfa, P. Druschel, and D. S. Wallach. Performance Analysis of TLS Web Servers. In *Network and Distributed System Security Symposium (NDSS)*, 2002.

[30] D. L. Cook, J. Ioannidis, A. D. Keromytis1, and J. Luck. CryptoGraphics: Secret Key Cryptography Using Graphics Cards . In *RSA Conference, Cryptographers Track (CT-RSA)*, 2005.

[31] N. Costigan and M. Scott. Accelerating SSL using the Vector processors in IBMs Cell Broadband Engine for Sonys Playstation 3. In *Cryptology ePrint Archive, Report*, 2007.

[32] J. Daemen and V. Rijmen. AES Proposal: Rijndael. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf, 1999.

[33] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[34] S. Dussé and B. Kaliski. A cryptographic library for the Motorola DSP56000. In *Advances in Cryptology—EUROCRYPT 1990*.

[35] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *ACM SIGCOMM*, 2010.

[36] O. Harrison and J. Waldron. Practical Symmetric Key Cryptography on Modern Graphics Hardware. In *USENIX Security Symposium*, 2008.

[37] O. Harrison and J. Waldron. Efficient Acceleration of Asymmetric Cryptography on Graphics Hardware. In *International Conference on Cryptology in Africa*, 2009.

[38] J. Jonsson and B. Kaliski. Public-key cryptography standards (PKCS) #1: RSA cryptography specifications version 2.1, 2003.

[39] E. Kasper and P. Schwabe. Faster and timing-attack resistant aes-gcm. In *Cryptographic Hardware and Embedded Systems (CHES)*. 2009.

[40] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel montgomery multiplication. In *Advances in Cryptology—EUROCRYPT 2000*, pages 523–538. Springer, 2000.

[41] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3th edition, 1997.

[42] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[43] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the internet. *SIGCOMM Comput. Commun. Rev.*, 40(4):135–146, 2010.

[44] S. A. Manavski. CUDA compatible gpu as an efficient hardware accelerator for aes cryptography.

[45] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[46] National Institute of Standards and Technology (NIST). *Recommendation for Key Management Part 1: General (Revised)*. 2007.

[47] NVIDIA Corp. *NVIDIA CUDA: Best Practices Guide, Version 3.1*. 2010.

[48] NVIDIA Corp. *NVIDIA CUDA: Programming Guide, Version 3.1*. 2010.

[49] D. A. Osvik, J. W. Bos, D. Stefan, and D. Canright. Fast software aes encryption. In *Foundations of Software Engineering (FSE)*, 2010.

[50] H. Park, K. Park, and Y. Cho. Analysis of the variable length nonzero window method for exponentiation. *Computers & Mathematics with Applications*, 37(7):21–29, 1999.

[51] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.

[52] E. Rescorla, A. Cain, and B. Korver. SSLACC: A Clustered SSL Accelerator. In *USENIX Security Symposium*, 2002.

[53] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[54] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG)*, 27(3):1–15, 2008.

[55] H. Shacham and D. Boneh. Improving SSL Handshake Performance via Batching. In *RSA Conference*, 2001.

[56] R. Szerwinski and T. Gneysu. Exploiting the Power of GPUs for Asymmetric Cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, 2008.

[57] S. B. 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 *USENIX OSDI*, 2008.

[58] J. Yang and J. Goodman. Symmetric Key Cryptography on Modern Graphics Hardware. In *ASIACRYPT*, 2007.

ServerSwitch: A Programmable and High Performance Platform for Data Center Networks

Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou[†*]
Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, Yongguang Zhang
Microsoft Research Asia, Beijing, China
[†] *Tsinghua University, Beijing, China*

Abstract

As one of the fundamental infrastructures for cloud computing, data center networks (DCN) have recently been studied extensively. We currently use pure software-based systems, FPGA based platforms, *e.g.*, NetFPGA, or OpenFlow switches, to implement and evaluate various DCN designs including topology design, control plane and routing, and congestion control. However, software-based approaches suffer from high CPU overhead and processing latency; FPGA based platforms are difficult to program and incur high cost; and OpenFlow focuses on control plane functions at present.

In this paper, we design a *ServerSwitch* to address the above problems. ServerSwitch is motivated by the observation that commodity Ethernet switching chips are becoming programmable and that the PCI-E interface provides high throughput and low latency between the server CPU and I/O subsystem. ServerSwitch uses a commodity switching chip for various customized packet forwarding, and leverages the server CPU for control and data plane packet processing, due to the low latency and high throughput between the switching chip and server CPU.

We have built our ServerSwitch at low cost. Our experiments demonstrate that ServerSwitch is fully programmable and achieves high performance. Specifically, we have implemented various forwarding schemes including source routing in hardware. Our in-network caching experiment showed high throughput and flexible data processing. Our QCN (Quantized Congestion Notification) implementation further demonstrated that ServerSwitch can react to network congestions in 23us.

[*] This work was performed when Zhiqiang Zhou was a visiting student at Microsoft Research Asia.

1 Introduction

Data centers have been built around the world for various cloud computing services. Servers in data centers are interconnected using data center networks. A large data center network may connect hundreds of thousands of servers. Due to the rise of cloud computing, data center networking (DCN) is becoming an important area of research. Many aspects of DCN, including topology design and routing [15, 5, 13, 11, 22], flow scheduling and congestion control [7, 6], virtualization [14], application support [26, 4], have been studied.

Since DCN is a relatively new exploration area, many of the designs (*e.g.*, [15, 5, 13, 22, 7, 14, 4]) have departed from the traditional Ethernet/IP/TCP based packet format, Internet-based single path routing (*e.g.*, OSPF), and TCP style congestion control. For example, Portland performs longest prefix matching (LPM) on destination MAC address, BCube advocates source routing, and QCN (Quantized Congestion Notification) [7] uses rate-based congestion control. Current Ethernet switches and IP routers therefore cannot be used to implement these designs.

To implement these designs, rich programmability is required. There are approaches that provide this programmability: pure software-based [17, 10, 16] or FPGA-based systems (*e.g.*, NetFPGA [23]). Software-based systems can provide full programmability and as recent progress [10, 16] has shown, may provide a reasonable packet forwarding rate. But their forwarding rate is still not comparable to commodity switching ASICs (application specific integrated circuit), and the batch processing used in their optimizations introduces high latency which is critical for various control plane functions such as signaling and congestion control [13, 22, 7]. Furthermore, the packet forwarding logics in DCN (*e.g.*, [15, 13, 22, 14]) are generally simple and hence are better implemented in silicon for cost and power savings. FPGA-based systems are fully programmable. But

the programmability is provided by hardware description languages such as Verilog, which are not as easy to learn and use as higher-level programming languages such as C/C++. Furthermore, FPGAs are expensive and are difficult to use in large volumes in data center environments.

In this paper, we design a ServerSwitch platform, which provides easy-to-use programmability, low latency and high throughput, and low cost. ServerSwitch is based on two observations as follows. First, we observe that commodity switching chips are becoming programmable. Though the programmability is not comparable to general-purpose CPUs, it is powerful enough to implement various packet forwarding schemes with different packet formats. Second, current standard PCI-E interface provides microsecond level latency and tens of Gb/s throughput between the I/O subsystem and server CPU. ServerSwitch is then a commodity server plus a commodity, programmable switching chip. These two components are connected via the PCI-E interface.

We have designed and implemented ServerSwitch. We have built a ServerSwitch card, which uses a merchandise gigabit Broadcom switching chip. The card connects to a commodity server using a PCI-E X4 interface. Each ServerSwitch card costs less than 400$ when manufactured in 100 pieces. We also have implemented a software stack, which manages the card, and provides support for control and data plane packet processing. We evaluated ServerSwitch using micro benchmarks and real DCN designs. We built a ServerSwitch based, 16-server BCube [13] testbed. We compared the performance of software-based packet forwarding and our ServerSwitch based forwarding. The results showed that ServerSwitch achieves high performance and zero CPU overhead for packet forwarding. We also implemented a QCN congestion control [7] using ServerSwitch. The experiments showed stable queue dynamics and that ServerSwitch can react to congestion in 23us.

ServerSwitch explores the design space of combining a high performance ASIC switching chip with limited programmability with a fully programmable multicore commodity server. Our key findings are as follows: 1) ServerSwitch shows that various packet forwarding schemes including source routing can be offloaded to the ASIC switching chip, hence resulting in small forwarding latency and zero CPU overhead. 2) With a low latency PCI-E interface, we can implement latency sensitive schemes such as QCN congestion control, using server CPU with a pure software approach. 3) The rich programmability and high performance provided by ServerSwitch can further enable new DCN services that need in-network data processing such as in-network caching [4].

The rest of the paper is organized as follows. We elaborate the design goals in § 2. We then present the architecture of ServerSwitch and our design choices in § 3. We illustrate the software, hardware, and API implementations in § 4. § 5 discusses how we use ServerSwitch to implement two real DCN designs, § 6 evaluates the platform with micro benchmarks and real DCN implementations. We discuss ServerSwitch limitations and 10G ServerSwitch in § 7. Finally, we present related work in § 8 and conclude in § 9.

2 Design Goals

As we have discussed in § 1, the goal of this paper is to design and implement a programmable and high performance DCN platform for existing and future DCN designs. Specifically, we have following design goals. First, on the data plane, the platform should provide a packet forwarding engine that is both programmable and achieves high-performance. Second, the platform needs to support new routing and signaling, flow/congestion control designs in the control plane. Third, the platform enables new DCN services (*e.g.*, in-network caching) by providing advanced in-network packet processing. To achieve these design goals, the platform needs to provide flexible programmability and high performance in both the data and control planes. It is highly desirable that the platform be easy to use and implemented in pure commodity and low cost silicon, which will ease the adoption of this platform in a real world product environment. We elaborate on these goals in detail in what follows.

Programmable packet forwarding engine. Packet forwarding is the basic service provided by a switch or router. Forwarding rate (packet per second, or PPS) is one of the most important metrics for network device evaluation. Current Ethernet switches and IP routers can offer line-rate forwarding for various packet sizes. However, recent DCN designs require a packet forwarding engine that goes beyond traditional destination MAC or IP address based forwarding. Many new DCN designs embed network topology information into server addresses and leverage this topology information for packet forwarding and routing. For example, PortLand [22] codes its fat-tree topology information into device MAC addresses and uses Longest Prefix Matching (LPM) over its PMAC (physical MAC) for packet forwarding. BCube uses source routing and introduces an NHI (Next Hop Index, §7.1 of [13]) to reduce routing path length by leveraging BCube structural information. We expect that more DCN architectures and topologies will appear in the near future. These new designs call for a programmable packet forwarding engine which can handle various packet forwarding schemes and packet formats.

New routing and signaling, flow/congestion control support. Besides the packet forwarding functions in the data plane, new DCN designs also introduce new control

and signaling protocols in the control plane. For example, to support the new addressing scheme, switches in PortLand need to intercept the ARP packets, and redirect them to a Fabric Manager, which then replies with the PMAC of the destination server. BCube uses adaptive routing. When a source server needs to communicate with a destination server, the source server sends probing packets to probe the available bandwidth of multiple edge-disjoint paths. It then selects the path with the highest available bandwidth. The recent proposed QCN switches sample the incoming packets and send back queue and congestion information to the source servers. The source servers then react to the congestion information by increasing or decreasing the sending rate. All these functionalities require the switches to be able to filter and process these new control plane messages. Control plane signaling is time critical and sensitive to latency. Hence switches have to process these control plane messages in real time. Note that current switches/routers do offer the ability to process the control plane messages with their embedded CPUs. However, their CPUs mainly focus on management functions and are generally lack of the ability to process packets with high throughput and low latency.

New DCN service support by enabling in-network packet processing. Unlike the Internet which consists of many ISPs owned by different organizations, data centers are owned and administrated by a single operator. Hence we expect that technology innovations will be adopted faster in the data center environment. One such innovation is to introduce more intelligence into data center networks by enabling in-network traffic processing. For example, CamCube [4] proposed a cache service by introducing packet filtering, processing, and caching in the network. We can also introduce switch-assisted reliable multicast [18, 8] in DCN, as discussed in [26]. For an in-network packet processing based DCN service, we need the programmability such as arbitrary packet modification, processing and caching, which is much more than the programmability provided by the programmable packet forwarding engine in our first design goal. More importantly, we need low overhead, line-rate data processing, which may reach several to tens of Gb/s.

The above design goals call for a platform which is programmable for both data and control planes, and it needs to achieve high throughput and low processing latency. Besides the programmability and high performance design goals, we have two additional requirements (or constraints) from the real world. First, the programmability we provide should be easy to use. Second, it is highly desirable that the platform is built from (inexpensive) commodity components (*e.g.*, merchandise chips). We believe that a platform based on commodity components has a pricing advantage over non-commodity, expensive ones. The easy-to-program requirement ensures the platform is easy to use, and the commodity constraint ensures the platform is amenable to wide adoption.

Our study revealed that none of the existing platforms meet all our design goals and the easy-to-program and commodity constraints. The pure software based approaches, *e.g.*, Click, have full and easy-to-use programmability, but cannot provide low latency packet processing and high packet forwarding rate. FPGA-based systems, *e.g.*, NetFPGA, are not as easy to program as the commodity servers, and their prices are generally high. For example, the price of Virtex-II Pro 50 used in NetFPGA is 1,180$ per chip for 100+ chip quantum listed on the Xilinx website. Openflow switches provide certain programmability for both forwarding and control functions. But due to the separation of switches and the controller, it is unclear how Openflow can be extended to support congestion control and in-network data processing.

We design ServerSwitch to meet the three design goals and the two practical constraints. ServerSwitch has a hardware part and a software part. The hardware part is a merchandise switching chip based NIC plus a commodity server. The ServerSwitch software manages the hardware and provides APIs for developers to program and control ServerSwitch. In the next section, we will describe the architecture of ServerSwitch, and how ServerSwitch meets the design goals and constraints.

3 Design

3.1 ServerSwitch Architecture

Our ServerSwitch architecture is influenced by progress and trends in ASIC switching chip and server technologies. First, though commodity switches are black boxes to their users, the switching chips inside (*e.g.*, from Broadcom, Fulcrum, and Marvell) are becoming increasingly programmable. They generally provide exact matching (EM) based on MAC addresses or MPLS tags, provide longest prefix matching (LPM) based on IP addresses, and have a TCAM (ternary content-addressable memory) table. Using this TCAM table, they can provide arbitrary field matching. Of course, the width of the arbitrary field is limited by the hardware, but is generally large enough for our purpose. For example, Broadcom Enduro series chips have a maximum width of 32 bytes, and Fulcrum FM3000 can match up to 78 bytes in the packet header [3]. Based on the matching result, the matched packets can then be programmed to be forwarded, discarded, duplicated (*e.g.*, for multicast purpose), or mirrored. Though the programmability is limited, we will show later that it is already enough for

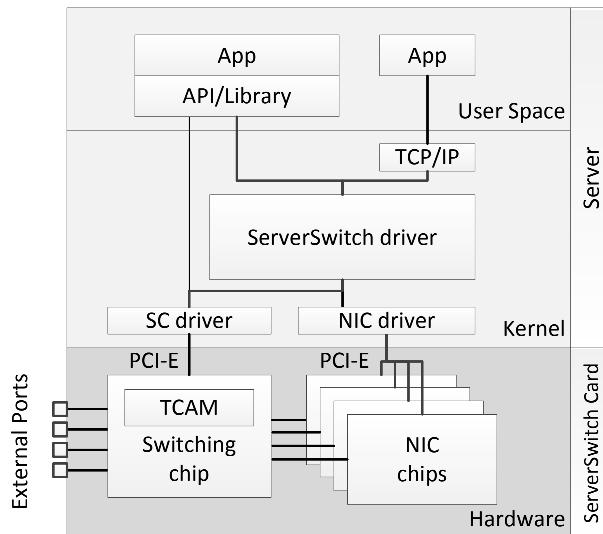


Figure 1: ServerSwitch architecture.

all packet forwarding functions in existing, and arguably, many future DCN designs.

Second, commodity CPU (*e.g.*, x86 and X64 CPUs) based servers now have a high-speed, low latency interface, *i.e.*, PCI-E, to connect to I/O subsystems such as a network interface card (NIC). Even PCI-E 1.0 X4 can provide 20Gbps bidirectional throughput and microsecond latency between the server CPU and NIC. Moreover, commodity servers are arguably the best programmable devices we currently have. It is very easy to write kernel drivers and user applications for packet processing with various development tools (*e.g.*, C/C++).

ServerSwitch then takes advantage of both commodity servers and merchandise switching chips to meet our design goals. Fig. 1 shows its architecture. The hardware part is an ASIC switching chip based NIC and a commodity server. The NIC and server are connected by PCI-E. From the figure, we can see there are two PCI-E channels. One is for the server to control and program the switching chip, the other is for data packet exchange between the server and switching chip.

The software part has a kernel and an application component, respectively. The kernel component has a switching chip (SC) driver to manage the commodity switching chip and an NIC driver for the NICs. The central part of the kernel component is a ServerSwitch driver, which sends and receives control messages and data packets through the SC and NIC drivers. The ServerSwitch driver is the place for various control messages, routing, congestion control, and various in-network packet processing. The application component is for developers. Developers use the provided APIs to interface with the ServerSwitch driver, and to program and control the switching chip.

Our ServerSwitch nicely fulfills all our design goals and meets the easy-to-program and commodity constraints. The switching chip provides a programmable packet forwarding engine which can perform packet matching based on flexible packet fields, and achieve full line rate forwarding even for small packet sizes. The ServerSwitch driver together with the PCI-E interface achieves low latency communication between the switching chip and server CPU. Hence various routing, signaling and flow/congestion controls can be well supported. Furthermore, the switch chip can be programmed to select specific packets into the server CPU for advanced processing (such as in-network caching) with high throughput. The commodity constraint is directly met since we use only commodity, inexpensive components in ServerSwitch. ServerSwitch is easy to use since all programming is performed using standard C/C++. When a developer introduces a new DCN design, he or she needs only to write an application to program the switching chip, and add any needed functions in the ServerSwitch driver.

The ability of our ServerSwitch is constrained by the abilities of the switching chip, the PCI-E interface, and the server system. For example, we may not be able to handle packet fields which are beyond the TCAM width, and we cannot further cut the latency between the switching chip and server CPU. In practice, however, we are still able to meet our design goals with these constraints. In the rest of this section, we will introduce the programmable packet forwarding engine, the software, and the APIs in detail.

3.2 ASIC-based Programmable Packet Forwarding Engine

In this section, we discuss how existing Ethernet switching chips can be programmed to support various packet forwarding schemes.

There are three commonly used forwarding schemes in current DCN designs, *i.e.*, Destination Address (DA) based, tag-based, and Source Routing (SR) based forwarding. DA-based forwarding is widely adopted by Ethernet and IP networks. Tag-based forwarding decouples routing from forwarding which makes traffic engineering easier. SR-based forwarding gives the source server ultimate control of the forwarding path and simplifies the functions in forwarding devices. Table 1 summarizes the forwarding primitives and existing DCN designs for these three forwarding schemes. There are three basic primitives to forward a packet, *i.e.*, lookup key extraction, key matching, and header modification. Note that the matching criteria is independent of the forwarding schemes, *i.e.*, a forwarding scheme can use any matching criteria. In practice, two commonly used cri-

Scheme	Primitives			DCN Design
	Extract	Match	Modify	
DA-based	Direct	Any	No	Portland DCell
Tag-based	Direct	Any	SWAP/ POP/ PUSH	-
SR-based	Direct	Any	POP	VL2
	Indirect	Any	Change Index	BCube

Table 1: Forwarding schemes and primitives.

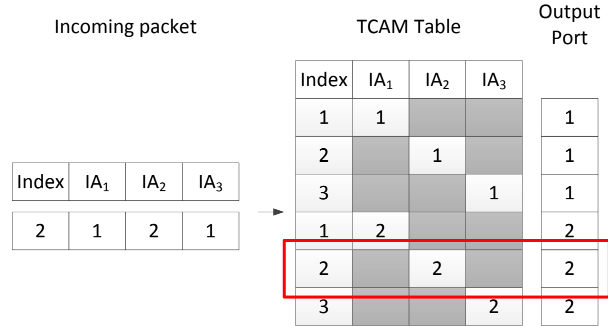


Figure 2: Support indexed source routing using TCAM.

teria are EM and LPM. Next, we describe the three forwarding schemes in detail. We start from SR-based forwarding.

3.2.1 Source Routing based Forwarding using TCAM

For SR-based forwarding, there are two approaches depending on how the lookup key is extracted: indexed and non-indexed SR-based forwarding. In both approaches, the source fills a series of intermediate addresses (IA) in the packet header to define the packet forwarding path. For the non-Indexed Source Routing (nISR), the forwarding engine always uses the first IA for table lookup and pops it before sending the packet. For Indexed Source Routing (ISR), there is an index i to denote the current hop. The engine first reads the index, then extracts IA_i based on the index, and finally updates the index before sending the packet. We focus on ISR support in the rest of this subsection. We will discuss nISR support in the next subsection since it can be implemented as a form of tag-based forwarding.

ISR-based forwarding uses two steps for lookup key extraction. It first gets the index from a fixed location, and then extracts the key pointed by the index. However, commodity switching chips rarely have the logic to perform this two-step indirect lookup key extraction. In this paper, we design a novel solution by leveraging TCAM and turning this two-step key extraction into a single step key extraction. The TCAM table has many entries and each entry has a value and a mask. The mask is to set the masking bits (‘care’ and ‘do-not-care’ bits) for the value.

In our design, for each incoming packet, the forward engine compares its index field and all IA fields against the TCAM table. The TCAM table is set up as follows. For each TCAM entry, the index field (i) and the IA_i field pointed by the index are ‘care’ fields. All other IA fields are ‘do-not-care’ fields. Thus, a TCAM entry can simultaneously match both the index and the corresponding IA_i field. As both index and IA_i may vary, we enumerate all the possible combinations of index and IA values in the TCAM table. When a packet comes in, it will match one and only one TCAM entry. The action of that entry determines the operation on that matched packet.

Fig. 2 illustrates how the procedure works. The incoming packet has one index field and three IA fields. IA_2 is the lookup key for this hop. In the TCAM table, the white fields are the ‘care’ fields and the gray fields are the ‘do-not-care’ fields. Suppose there are two possible IA addresses and the maximum value of the index is three, there are 6 entries in the TCAM table. For this incoming packet, it matches the 5th entry where Index=2 and IA_2 = 2. The chip then directs the packet to output port 2. In § 5.1, we will describe the exact packet format based on our ServerSwitch.

This design makes a trade-off between the requirement of extra ASIC logic and the TCAM space. When there are n different IA values, the two-step indirect matching method uses n lookup entries, while this one-step method uses $n \times d$ entries where d is the maximum value of the index. d is always less than or equal to the network diameter. Modern switching chips have at least thousands of TCAM entries, so this one-step method works well in the DCN environment. For example, consider a medium sized DCN such as a three-level fat-tree in Portland. When using 48-port switches, there are 27,648 hosts. We can use 48 IA values to differentiate these 48 next hop ports. Since the diameter of the network is 6, the number of TCAM entries is 48×6 = 288, which is much smaller than the TCAM table size.

3.2.2 Destination and Tag-based Forwarding

As for the DA-based forwarding, the position of the lookup key is fixed in the packet header and the forwarding engine reads the key *directly* from the packet header. No lookup key modification is needed since the destination address is a globally unique id. However, the destination address can be placed anywhere in the packet header, so the engine must be able to perform matching on arbitrary fields. For example, Portland requires the switch to perform LPM on the destination MAC address,

whereas DCell uses a self-defined header.

Tag-based routing also uses direct key extraction, but the tag needs to be modified on a per-hop basis since the tags have only local meaning. To support this routing scheme, the forwarding engine must support SWAP/POP/PUSH operations on tags.

Modern merchandise switching chips generally have a programmable parser, which can be used to extract arbitrary fields. The TCAM matching module is flexible enough to implement EM, LPM [25], and range matching. Hence, DA-based forwarding can be well supported.

For tag-based forwarding, many commodity switching chips for Metro Ethernet Network already support MPLS (multiple protocol label switching), which is the representative tag-based forwarding technology. Those chips support POP/PUSH/SWAP operations on the MPLS labels in the packet header. Hence we can support tag-based forwarding by selecting a switching chip with MPLS support. Further, by using tag stacking and POP operations, we can also support nISR-based forwarding. In such nISR design, the source fills a stack of tags to denote the routing path and the intermediate switches use the outermost tag for table lookup and then pops the tag before forwarding the packet.

3.3 Server Software

3.3.1 Kernel Components

The ServerSwitch driver is the central hub that receives all incoming traffic from the underlying ServerSwitch card. The driver can process them itself or it can deliver them to the user space for further processing. Processing them in the driver gives higher performance but requires more effort to program and debug. Meanwhile, processing these packets in user space is easy for development but scarifies performance. Instead of making a choice on behalf of users, ServerSwitch allows users to decide which one to use. For low rate control plane traffic where processing performance is not a major concern, *e.g.*, ARP packets, ServerSwitch can deliver them to user space for applications to process them. Since the applications need to send control plane traffic too, ServerSwitch provides APIs to receive packets from user-space applications to be sent down to the NIC chips. For those control plane packets with low latency requirement and high speed in-network processing traffic whose performance is a major concern, *e.g.*, QCN queue queries or data cache traffic, we can process them in the ServerSwitch driver.

The SC and NIC drivers both act as the data channels between the switching chip and the ServerSwitch driver. They receive packets from the device and deliver them to the ServerSwitch driver, and vice versa. The SC driver also provides an interface for the user library and the ServerSwitch to manipulate its registers directly, so both applications and the ServerSwitch driver can control the switching chip directly.

3.3.2 APIs

We design a set of APIs to control the switching chip and send/receive packets. The APIs include five categories as follows.

1. Set User Defined Lookup Key (UDLK): This API configures the programmable parser in the switching chip by setting the *i*-th UDLK. In this API, the UDLK can be fields from the packet header as well as meta-data, *e.g.*, the incoming port of a packet. We use the most generic form to define packet header fields, *i.e.*, the byte position of the desired fields. In the following example, we set the destination MAC address (6 bytes, B0-5) as the first UDLK. We can also combine meta-data (*e.g.*, incoming port) and non-consecutive byte range to define a UDLK, as shown in the second statement which is used for BCube (§ 5.1).

```
API:
  SetUDLK(int i, UDLK udlk)
Example:
  SetUDLK(1, (B0-5))
  SetUDLK(2, (INPORT, B30-33, B42-45))
```

2. Set Lookup Table: There are several lookup tables in the switching chip, a general purpose TCAM table, and protocol specific lookup tables for Ethernet, IP, and MPLS. This API configures different lookup tables denoted by *type*, and sets the *value*, *mask* and *action* for the *i*-th entry. The mask is NULL when the lookup table is an EM table. The *action* is a structure that defines the actions to be taken for the matched packets, *e.g.*, directing the packets to a specified output port, performing pre-defined header modifications, etc. For example, for MPLS the modification actions can be Swap/Pop/Push. The *iudlk* is the index of UDLK to be compared. *iudlk* is ignored for the tables that do not support UDLK.

In the following example, the statement sets the first TCAM entry and compares the destination MAC address (the first UDLK) with the value field (000001020001, *i.e.*, 00:00:01:02:00:01) using mask (FFFFFF000000). This statement is used to perform LPM on dest MAC for PortLand. Consequently, all matching packets are forwarded to the third virtual interface.

```
API:
  SetLookupTable(int type, int i,
    int iudlk, char *value, char* mask,
    ACTION *action)
Example:
  SetLookupTable(TCAM, 1,
    1, "000001020001", "FFFFFF000000",
    {act=REDIRECT_VIF, vif=3})
```

3. Set Virtual Interface Table: This API sets up the *i*-th virtual interface entry which contains destination and source MAC addresses as well as the output port. The MAC addresses are used to replace the original MACs in the packet when they are not NULL.

For example, the following command sets up the third virtual interface to deliver packets to output port 2. Meanwhile, the destination MAC is changed to the given value (001F29D417E8) accordingly. The edge switches in Portland need such functionality to change PMAC back to the original MAC (§3.2 in [22]).

```
API:
  SetVIfTable(int i, char *dmac,
        char *smac, int oport)
Example:
  SetVIfTable(3, "001F29D417E8", NULL, 2)
```

4. Read/Write Registers: There are many statistic registers in switching chip, *e.g.*, queue length and packet counters, and registers to configure the behaviors of the switching chip, *e.g.*, enable/disable L3 processing. This API is to read and write those registers (specified by *regname*). As an example, the following command returns the queue length (in bytes) of output port 0.

```
API:
  int ReadRegister (int regname)
  int WriteRegister(int regname, int value)
Example:
  ReadRegister(OUTPUT_QUEUE_BYTES_PORT0)
```

5. Send/Receive Packet: There are multiple NICs for sending and receiving packets. We can use the first API to send packet to a specific NIC port (*oport*). When we receive a packet, the second API also provides the input NIC port (*iport*) for the packet.

```
API:
  int SendPacket(char *pkt, int oport)
  int RecvPacket(char *pkt, int *iport)
```

4 Implementation

4.1 ServerSwitch Card

Fig. 3 shows the ServerSwitch card we designed. All chips used on the card are merchandise ASICs. The Broadcom switching chip BCM56338 has 8 Gigabit Ethernet (GE) ports and two 10GE ports [1]. Four of the GE ports connect externally and the other four GE ports connect to two dual GE port Intel 82576EB NIC chips. The two NIC chips are used to carry a maximum of 4Gb/s traffic between the switching chip and the server since the bandwidth of the PCI-E interface on 56338 is only 2Gb/s. The three chips connect to the server via

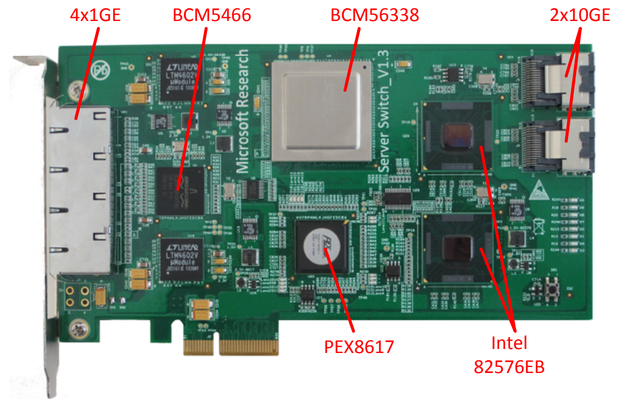


Figure 3: ServerSwitch card.

a PCI-E switch PLX PEX8617. The effective bandwidth from the PEX8617 to BCM56338, the two NIC chips and the server are 2, 8, 8 and 8Gb/s (single direction). Since the maximum inbound or outbound traffic is 4Gb/s, PCI-E is not the bottleneck. The two 10GE XAUI ports are designed for interconnecting multiple ServerSwitch cards in one server chassis to create a larger non-blocking switching fabric with more ports. Each ServerSwitch card costs less than 400$ when manufactured in 100 pieces. We expect the price can be cut to 200$ for a quantity of 10K. The power consumption of ServerSwitch is 15.4W when all 8 GE ports are idle, and is 15.7W when all of them carry full speed traffic.

Fig. 4 shows the packet processing pipeline of the switching chip, which has three stages. First, when the packets go into the switching chip, they are directed to a programmable parser and a classifier. The classifier then directs the packets to one of the protocol specific header parsers. The Ethernet parser extracts the destination MAC address (DMAC), the IP parser extracts the destination IP address (DIP), the MPLS parser extracts the MPLS label and the Prog parser can generate two different UDLKs. Each UDLK can contain any aligned four 4-byte blocks from the first 128 bytes of the packet, and some meta-data of the packet.

Next, the DMAC is sent to the EM(MAC) matching module, the DIP to both the LPM and EM(IP) matching modules, the MPLS label to the EM(MPLS) module, and the UDLK to the TCAM. Each TCAM entry can select one of the two UDLKs to match. The matchings are performed in parallel. The three matching modules (EM, LPM, TCAM) result in an index into the interface table, which contains the output port, destination and source MAC. When multiple lookup modules match, the priority of their results follows TCAM > EM > LPM.

Finally, the packet header is modified by the L3 and L2 modifiers accordingly. The L3 modifier changes the L3 header, *e.g.*, IP TTL, IP checksum and MPLS label. The

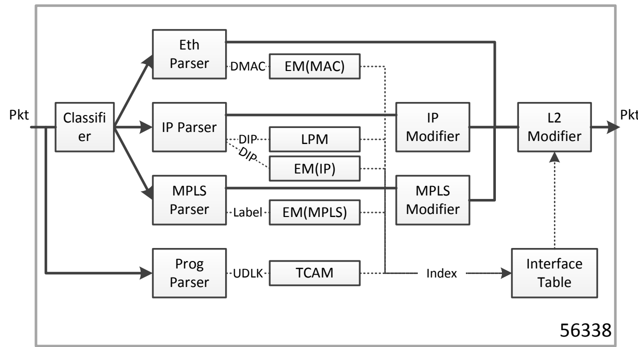


Figure 4: Packet processing pipeline in Broadcom 56338.

	Byte 1	Byte 2	Byte 3	Byte 4
B14-17	Version / HL	Tos	Total length	
B18-21	Identification		Flags / Fragment offset	
B22-25	TTL	Protocol	Header checksum	
B26-29	Source Address			
B30-33	Destination Address			
B34-37	NHA_1	NHA_2	NHA_3	NHA_4
B38-41	NHA_5	NHA_6	NHA_7	NHA_8
B42-45	BCube Protocol	NH	Pad	

Figure 5: BCube header on the ServerSwitch platform.

L2 modifier can use the MAC addresses in the interface table to replace the original MAC addresses.

The size of EM tables for MAC, IPv4 and MPLS are 32K, 8K and 4K entries, respectively. The LPM for IPv4 and the TCAM table have 6144 and 2K entries, respectively. The interface table has 4K entries. All these tables, the Prog Parser and the behaviors of the modifiers are programmable.

4.2 Kernel Drivers

We have developed ServerSwitch kernel drivers for Windows Server 2008 R2. As shown in Fig. 1, it has components as follows.

Switching Chip Driver. We implemented a PCI-E driver based on Broadcom's Dev Kits. The driver has 2670 lines of C code. It allocates a DMA region and maps the chip's registers into memory address using memory-mapped I/O (MMIO). The driver can deliver received packets to the ServerSwitch driver, and send packets to hardware. The ServerSwitch driver and user library can access the registers and thus control the switching chip via this SC driver.

NIC Driver. We directly use the most recent Intel NIC driver binaries.

ServerSwitch Driver. We implemented the ServerSwitch driver as a Windows NDIS MUX driver. It has 20719 lines of C code. The driver exports itself as a virtual NIC. It binds the TCP/IP stack on its top and the Intel NIC driver and the SC driver at its bottom. The driver uses `IRP` to send and receive packets from the user library. It can also deliver the packets to the TCP/IP stack. The ServerSwitch driver provides a kernel framework for developing various DCN designs.

4.3 User Library

The library is based on the Broadcom SDK. The SDK has 3000K+ lines of C code and runs only on Linux and VxWorks. We ported this SDK to Subsystem for UNIX-based Applications (SUA) on Windows Server 2008 [2]. At the bottom of the SDK, we added a library to interact with our kernel driver. We then developed ServerSwitch APIs over the SDK.

5 Building with the ServerSwitch Platform

In this section, we use ServerSwitch to implement several representative DCN designs. First, we implement BCube to illustrate how indexed source routing is supported in the switching chip of ServerSwitch. In our BCube implementation, BCube packet forwarding is purely carried out in hardware. Second, we show our implementation of QCN congestion control. Our QCN implementation demonstrates that our ServerSwitch can generate low latency control messages using the server CPU. Due to space limitation, we discuss how ServerSwitch can support other DCN designs in our technical report [19].

5.1 BCube

BCube is a server centric DCN architecture [13]. BCube uses adaptive source routing. Source servers probe multiple paths and select the one with the highest available bandwidth. BCube defines two types of control messages, for neighbor discovery (ND) and available bandwidth query (ABQ) respectively. The first one is for servers to maintain the forwarding table. The second one is used to probe the available bandwidth of the multiple parallel paths between the source and destination.

Our ServerSwitch is an ideal platform for implementing BCube. For an intermediate server in BCube, our ServerSwitch card can offload packet forwarding from the server CPU. For source and destination servers, our ServerSwitch card can achieve k:1 speedup using k NICs connected by BCube topology. This is because in our design the internal bandwidth between the server and the NICs is equal to the external bandwidth provided by the multiple NICs, as we show in Figure 1.

Fig. 5 shows the BCube header we use. It consists of an IP header and a private header (gray fields). We use this private header to implement the BCube header. We use an officially unassigned IP protocol number to differentiate the packet from normal TCP/UDP packets. In the private header, the BCube protocol is used to identify control plane messages. NH is the number of valid NHA fields. It is used by a receiver to construct a reverse path to the sender. There are 8 1-byte Next Hop Address (NHA) fields, defined in BCube for indexed source routing. Different from NHA in the original BCube header design, NHAs are filled in reverse order in our private header. NHA_1 is now the lookup key for the last hop. This implementation adaption is to obtain an automatic index counter by the hardware. We observe that for a normal IP packet, its TTL is automatically decreased after one hop. Therefore, we overload the TTL field in the IP header as the index field for NHAs. This is the reason why we store NHAs in reverse order.

We implemented a BCube kernel module in the ServerSwitch driver and a BCube agent at the user-level. The kernel module implements data plane functionalities. On the receiving direction, it delivers all received control messages to the user-level agent for processing. For any received data packets, it removes their BCube headers and delivers them to the TCP/IP stack. On the sending direction, it adds the BCube header for the packets from the TCP/IP stack and sends them to the NICs.

The BCube agent implements all control plane functionalities. It first sets up the ISR-based forwarding rules and the packet filter rules in the switching chip. Then, it processes the control messages. When it receives an ND message, it updates the interface table using `SetVIfTable`. It periodically uses `ReadRegister` to obtain traffic volume from the switching chip and calculates the available bandwidth for each port. When it receives an ABQ message, it encodes the available bandwidth in the ABQ message, and sends it to the next hop.

Fig. 6 shows the procedure to initialize the switching chip for BCube, using the ServerSwitch API. Line 1 sets a 12-byte $UDLK_1$ for source routing, including TTL (`B22`) and the NHA fields (`B34-41`). Line 2 sets another 9-byte $UDLK_2$ for packet filtering, including incoming port number (`INPORT`), IP destination address (`B30-33`) and BCube protocol (`B42`). The `INPORT` occupies 1-byte field. Lines 5-18 set the ISR-based TCAM table. Since every NHA denotes a neighbor node with a destination MAC and corresponding output port, line 8 sets up one interface entry for one NHA value. Lines 13-16 sets up a TCAM entry to match the TTL and its corresponding NHA in $UDLK_1$. Since the switch discards the IP packets whose TTL ≤ 1, we use TTL = 2 to denote NHA_1. Lines 21-38 filter packets to the server. Since the switching chip has four external (0-3) and four internal ports (4-7), we filter the traffic of an external port to a corresponding internal port, *i.e.*, port 0$\rightarrow$4, 1$\rightarrow$5, 2$\rightarrow$6 and 3$\rightarrow$7. Line 23 sets `action` to direct the packets to port $4 \sim 7$ respectively. Lines 27-29 match those packets whose destination BCube address equals the local BCube address in $UDLK_2$. Lines 32-37 match BCube control plane messages, *i.e.*, ND and ABQ, in $UDLK_2$. In our switching chip, when a packet matches multiple TCAM entries, the entry with the highest index will win. Therefore, in our BCube implementation, entries for control plane messages have higher priority than the other ones.

```
1:  SetUDLF(1, (B22-25, B34-41));
2:  SetUDLF(2, (INPORT, B30-33, B42-45));
3:
4:  // setup ISR-based forwarding table
5:  j = 0;
6:  foreach nha in (all possible NHA values)
7:  {
8:    SetVifTable(nha, dstmac, srcmac, oport);
9:    for (index = 0; index < 8; index++)
10:   {
11:     // val[0] matches B22 (TTL) in UDLF_1
12:     // val[4:11] matches B34-41 (NHAs) in UDLF_1
13:     val[0]   = index+2; mask[0] = 0xff;
14:     val[4+index] = nha; mask[4+index] = 0xff;
15:     action.act = REDIRECT_VIF;  action.vif = nha;
16:     SetLookupTable(TCAM,j++,1,val,mask,&action);
17:   }
18: }
19:
20: // setup filter to server
21: for (i = 0; i < 4; i++)
22: {
23:   action.act = REDIRECT_PORT; action.port = 4 + i
24:   // filter packets that are sent to localhost
25:   // val[0] matches INPORT in UDLF_2
26:   // val[1:4] match B30-33 (IP dst addr) in UDLF_2
27:   val[0]   = i; mask[0] = 0xff
28:   val[1:4] = my_bcube_id; mask[1:4] = 0xffffffff;
29:   SetLookupTable(TCAM,j++,2,val,mask,&action);
30:   // filter control plane packets
31:   // val[5] matches B42 (BCube prot) in UDLF_2
32:   val[0] = i;   mask[0] = 0xff;
33:   val[5] = ND;  mask[5] = 0xff;
34:   SetLookupTable(TCAM,j++,2,val,mask,&action);
35:   val[0] = i;   mask[0] = 0xff;
36:   val[5] = ABQ; mask[5] = 0xff;
37:   SetLookupTable(TCAM,j++,2,val,mask,&action);
38: }
```

Figure 6: Pseudo TCAM setup code for BCube.

5.2 Quantized Congestion Control (QCN)

QCN is a rate-based congestion control algorithm for the Ethernet environment [7]. The algorithm has two parts. The Switch or Congestion Point (CP) adaptively samples incoming packets and generates feedback messages addressed to the source of the sampled packets. The feedback message contains congestion information at the CP. The Source or Reaction Point (RP) then reacts based on the feedback from the CP. See [7] for QCN details. The previous studies of QCN are based on simulation or hardware implementation.

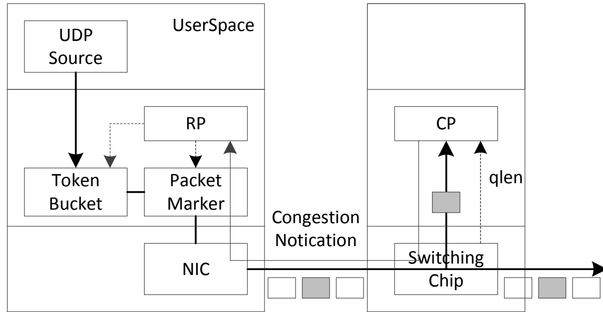


Figure 7: QCN on the ServerSwitch platform.

We implemented QCN on the ServerSwitch platform as shown in Fig. 7. The switching chip we use cannot adaptively sample packets based on the queue length, so we let the source mark packets adaptively and let the ServerSwitch switching chip mirror the marked packets to the ServerSwitch CPU. When the ServerSwitch CPU receives the marked packets, it immediately reads the queue length from the switching chip and sends the Congestion Notification (CN) back to the source. When the source receives the CN message, it adjusts its sending rate and marking probability.

We implemented the CP and RP algorithms in ServerSwitch and end-host respectively based on the most recent QCN Pseudo code V2.3 [24]. In order to minimize the response delay, the CP module is implemented in the ServerSwitch driver. The CP module sets up a TCAM entry to filter marked packets to the CPU. On the end-host, we implemented a token bucket rate limiter in the kernel to control the traffic sending rate at the source.

6 Evaluation

Our evaluation has two parts. In the first part, we show micro benchmarks for our ServerSwitch. We evaluate its performance on packet forwarding, register read/write, and in-network caching. For micro benchmark evaluation, we connect our ServerSwitch to a NetFPGA card and use NetFPGA to generate traffic. In the second part, we implement two DCN designs, namely BCube and QCN, using ServerSwitch. We build a 16-server BCube_1 network to run BCube and QCN experiments. We currently build only two ServerSwitch cards. As shown in Fig. 8, the two gray nodes are equipped with ServerSwitch cards, they use an ASUS motherboard with Intel Quad Core i7 2.8GHz CPU. The other 14 servers are Dell Optiplex 755 with 2.4Ghz dual core CPU. The switches are 8-port DLink DGS-1008D GE switches.

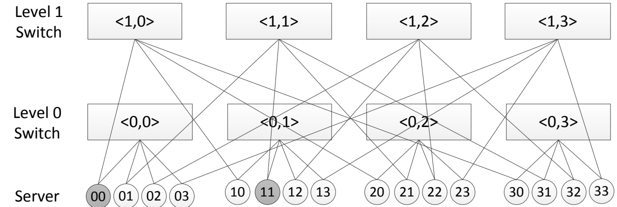


Figure 8: BCube_1 testbed.

6.1 Micro Benchmarks

We directly connect the four GE ports of one ServerSwitch to the four GE ports of one NetFPGA, and use the NetFPGA-based packet generator to generate line-rate traffic to evaluate the packet forwarding performance of ServerSwitch. We record the packet send and receive time using NetFPGA to measure the forwarding latency of ServerSwitch. The precision of the timestamp recorded by NetFPGA is $8ns$.

Forwarding Performance. Fig. 9 compares the forwarding performance of our ServerSwitch card and a software-based BCube implementation using an ASUS quad core server. In the evaluation, we use NetFPGA to generate 4GE traffic. The software implementation of the BCube packet forwarding is very simple. It uses NHA as an index to get the output port. (See §7.2 in [13] for more details) As we can see, there is a very huge performance gap between these two approaches. For ServerSwitch, there is no packet drop for any packet sizes, and the forwarding delay is small. The delays for 64 bytes and 1514 bytes are 4.3us and 15.6us respectively, and it grows linearly with the packet size. The slope is 7.7ns per byte, which is very close to the transmission delay of one byte over a GE link. The curve suggests the forwarding delay is a 4.2us fixed processing delay plus the transmission delay. For software forwarding, the maximum PPS achieved is 1.73Mpps and packets get dropped when the packet size is less than or equal to 512 bytes. The CPU utilization for 1514 byte is already 65.6%. Moreover, the forwarding delay is also much larger than that of ServerSwitch. This experiment suggests that a switching chip does a much better job for packet forwarding, and that using software for 'simple' packet forwarding is not efficient.

Register Read/Write Performance. Certain applications need to read and write registers of the switching chip frequently. For example, our software-based QCN needs to frequently read queue length from the switching chip. In this test, we continuously read/write a 32-bit register 1,000,000 times, and the average R/W latency of one R/W operation is 6.94/4.61us. We note that the latency is larger than what has been reported before (around $1us$) [20]. This is because [20] measured

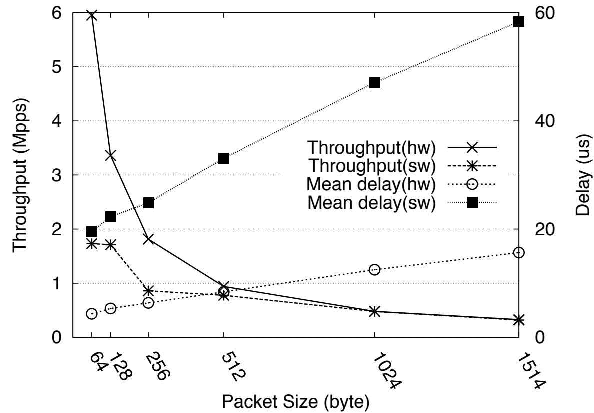


Figure 9: Packet forwarding performance.

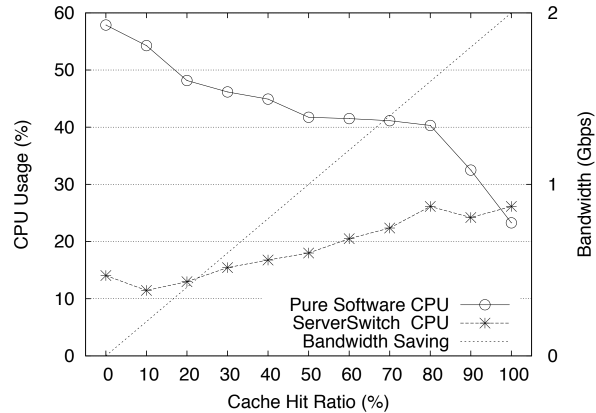


Figure 10: CPU utilization for in-network caching.

the latency of a single MMIO R/W operation, whereas our registers are not mapped but are accessed indirectly via several mapped registers. In our case, a read operation consists of four MMIO write and three MMIO read operations. We note that the transmission delay of one 1514-bytes packet over 1GE link is $12us$, so the read operation of our ServerSwitch can be finished within the transmission time of one packet.

In-network Caching Performance. We show that ServerSwitch can be used to support in-network caching. In this experiment, ServerSwitch uses two GbE ports to connect to NetFPGA A and the other two ports to NetFPGA B. NetFPGA A sends request packets to B via ServerSwitch. When B receives one request, it replies with one data packet. The sizes of request and reply are 128 and 1514 bytes, respectively. Every request or reply packet carries a unique ID in its packet header. When ServerSwitch receives a request from A, the switching chip performs an exact matching on the ID of the request. A match indicates that the ServerSwitch has already cached the response packet. The request is then forwarded to the server CPU which sends back the cached copy to A. When there is no match, the request is forwarded to B, and B sends back the response data. ServerSwitch also oversees the response data and tries to cache a local copy. The request rate per link is 85.8Mb/s, so the response rate per link between ServerSwitch and A is 966Mb/s. Since one NetFPGA has 4 ports, we use one NetFPGA to act as both A and B in the experiment.

We vary the cache hit ratio at ServerSwitch and measure the CPU overhead of the ServerSwitch. In-network caching increases CPU usage at ServerSwitch, but saves bandwidth between B and ServerSwitch. In our toy network setup, a $x\%$ cache hit rate directly results in $x\%$ bandwidth saving between B and ServerSwitch (as shown in Fig. 10). In a real network environment, we expect the savings will be more significant since we can save more bandwidth for multi-hop cases.

Fig. 10 also shows the CPU overhead of the ServerSwitch for different cache hit ratios. Of course, the higher the cache hit ratio, the more bandwidth we can save and the more CPU usage we need to pay. Note that in Fig. 10, even when the cache hit ratio is 0, we still have a cost of 14% CPU usage. This is because ServerSwitch needs to do caching for the 1.9Gbps response traffic from B to ServerSwitch. Fig. 10 also includes the CPU overhead of a pure software-based caching implementation. Our result clearly shows that our ServerSwitch significantly outperforms pure software-based caching.

6.2 ServerSwitch based BCube

In this experiment, we set up two TCP connections C1 and C2 between servers 01 and 10. The two connections use two parallel paths, P1 $\{01, 00, 10\}$ for C1 and P2 $\{01, 11, 10\}$ for C2, respectively. We run this experiment twice. First, we configure 00 and 11 to use the ServerSwitch cards for packet forwarding. Next, we configure them to use software forwarding. In both cases, the total throughput is 1.7Gbps and is split equally into the two parallel paths. When using ServerSwitch for forwarding, both 00 and 11 use zero CPU cycles. When using software forwarding, both servers use 15% CPU cycles. Since both servers have a quad core CPU, 15% CPU usage equals 60% for one core.

6.3 ServerSwitch based QCN

In this experiment, we configure server 00 to act as a QCN-enabled node. We use `iperf` to send UDP traffic from server 01 to 10 via 00. The sending rate of `iperf` is limited by the traffic shaper at 01. When there is congestion on level-1 port of 00, 00 sends CN to 01. We use the QCN baseline parameters [7] in this experiment.

Fig. 11 shows the throughput of the UDP traffic and the output queue length at server 00. When we start the UDP traffic, level 1 port is 1Gb/s. There is no congestion and the output queue length is zero. At time 20s, we limit level 1 port at 00 to 200Mb/s, the queue immediately builds up and causes 00 to send CN to the source. The source starts to use the QCN algorithm to adjust its traffic rate in order to maintain the queue length around Q_EQ which is 50KB in this experiment. We can see that the sending rate decreases to 200Mb/s very fast. And then we increase the bandwidth by 200Mb/s every 20 seconds. Similarly, the source adapts quickly to the new bandwidth. As shown in the figure, the queue length fluctuates around Q_EQ. This shows that this software-based implementation performs good congestion control. The rate of queue query packets processed by node 00 is very low during the experiment, with maximum and mean values of 801 and 173 pps. Hence QCN message processing introduces very little additional CPU overhead. The total CPU utilization is smaller than 5%. Besides, there is no packet drop in the experiment, even at the point when we decrease the bandwidth to 200Mb/s. QCN therefore achieves lossless packet delivery. We have varied the Q_EQ from 25KB to 200KB and the results are similar.

The extra delay introduced by our software approach to generate a QCN queue reply message consists of three parts: directing the QCN queue query to the CPU, reading the queue register, and sending back the QCN queue reply. To measure this delay, we first measure time RTT_1 between the QCN query and reply at 01. Then we configure the switching chip to simply bounce the QCN query back to the source assuming zero delay response for hardware implementation. We measure the time RTT_2 between sending and receiving a QCN query at 01. RTT_1 - RTT_2 reflects the extra delay introduced by software. The packet sizes of the queue query and reply are both 64 bytes in this measurement. The average values of RTT_1 and RTT_2 are 41us and 18us based on 10,000 measurements. Our software introduces only 23us delay. This extra delay is tolerable since it is comparable to or smaller than the packet transmission delay for one single 1500-bytes in a multi-hop environment.

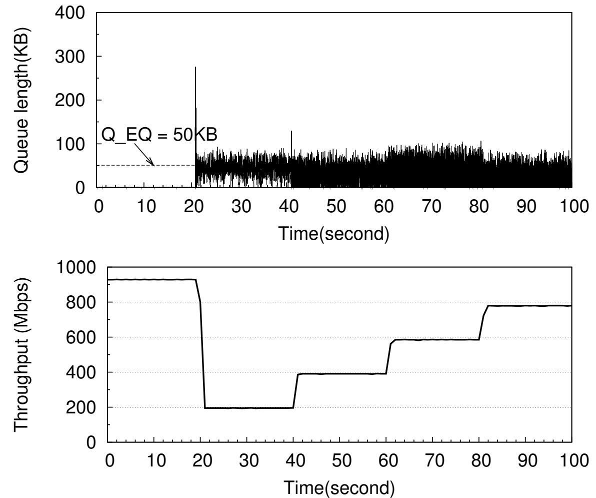


Figure 11: Throughput and queue dynamics during bandwidth change.

7 Discussion

Limitations of ServerSwitch. The current version of ServerSwitch has the following limitations: 1) Limited hardware forwarding programmability. The switching chip we use has limited programmability on header field modification. It supports only standard header modifications of supported protocols (*e.g.*, changing Ethernet MAC addresses, decreasing IP TTL, changing IP DSCP, adding/removing IP tunnel header, modifying MPLS header). Due to the hardware limitation, our implementation of index-based source routing has to re-interpret the IP TTL field. 2) Relatively high packet processing latency due to switching chip to CPU communication. For the packets that require 'real' per-packet processing such as congestion information calculation in XCP protocol, the switching chip must deliver them to the CPU for processing, which leads to higher latency. Hence ServerSwitch is not suitable for protocols that need real time per-packet processing such as XCP. 3) Restricted form factor and relatively low speed. At present, a ServerSwitch card provides only 4 GbE ports. Though it can be directly used for server-centric or hybrid designs, *e.g.*, BCube, DCell, and CamCube, we do not expect that the current ServerSwitch can be directly used for architectures that need a large number of switch ports (48-ports or more), *e.g.*, fat-tree and VL2. However, since 4 ServerSwitch cards can be connected together to provide 16 ports, we believe ServerSwitch is still a viable platform for system prototyping for such architectures.

10GE ServerSwitch. Using the same hardware architecture, we can build a 10GE ServerSwitch. We need to upgrade the Ethernet switching chip, the PCI-E switching chip and the NIC chips. As for the Ethernet switching chip, 10GbE switching chips with 24x10GbE ports or more are already available from Broadcom, Fulcrum or Marvell. We can use two dual 10GbE Ethernet controller chips to provide a 40Gb/s data channel between the card and server CPU. Since we do not expect all traffic to be delivered to the CPU for processing, the internal bandwidth between the card and the server does not need to match the total external bandwidth. In this case, the number of external 10GE ports can be larger than four. We also need to upgrade the PCI-E switching chip to provide

an upstream link with 40Gb/s bandwidth, which requires PCI-E Gen2 x8. Since the signal rate on the board is 10x faster than that in the current ServerSwitch, more hardware engineering effort will be needed to guarantee the Signal Integrity (SI).

All the chips discussed above are readily available in the market. The major cost of such a 10GbE card comes from the 10GbE Ethernet switching chip, which has a much higher price than the 8xGbE switching chip. For example, a chip with 24 10GbE ports may cost about 10x that of the current one. The NIC chip and PCI-E switching chip cost about 2x∼3x than current ones. Overall, we expect the 10GE version card to be about 5x more expensive than the current 1GE version.

8 Related Work

OpenFlow defines an architecture for a central controller to manage OpenFlow switches over a secure channel, usually via TCP/IP. It defines a specification to manage the flow table inside the switches. Both OpenFlow and ServerSwitch aim towards a more programmable networking platform. Aiming to provide both programmability and high performance, ServerSwitch uses multiple PCI-E lanes to interconnect the switching chip and the server. The low latency and high speed of the channel enables us to harness the resources in a commodity server to provide both programmable control and data planes. With Openflow, however, it is hard to achieve similar functionalities due to the higher latency and lower bandwidth between switches and the controller.

Orphal provides a common API for proprietary switching hardware [21], which is similar to our APIs. Specifically, they also designed a set of APIs to manage the TCAM table. Our work is more than API design. We introduce a novel TCAM table based method for index-based source routing. We also leverage the resources of a commodity server to provide extra programmability.

Flowstream uses commodity switches to direct traffic to commodity servers for in-network processing [12]. The switch and the server are loosely coupled, *i.e.*, the server cannot directly control the switching chip. In ServerSwitch, the server and the switching chip are tightly coupled, which enables ServerSwitch to provide new functions such as software-defined congestion control which requires low-latency communication between the server and the switching chip.

Recently, high performance software routers, *e.g.*, RouteBricks [10] and PacketShader [16] have been designed and implemented. By leveraging multi-cores, they can achieve tens of Gb/s throughput. ServerSwitch is complementary to these efforts in that ServerSwitch tries to offload certain packet forwarding tasks from the CPU to a modern switching chip. ServerSwitch also tries to optimize its software to process low latency packets such as congestion control messages. At present, due to hardware limitations, ServerSwitch only provides 4x1GE ports. RouteBricks or PacketShader can certainly leverage a future 10GE ServerSwitch card to provide a higher throughput system, with a portion of traffic forwarded by the switching chip.

Commercial switches generally have an embedded CPU for switch management. More recently, Arista's 7100 series introduces the use of dual-core x86 CPU and provides APIs for programmable management plane processing. ServerSwitch differs from existing commodity switches in two ways: (1) The CPUs in commodity switches mainly focus on management functions, whereas ServerSwitch explores a way to combine the switching chip with the most advanced CPUs and server architecture. On this platform, the CPUs can process forwarding/control/management plane packets with high throughput and low latency. The host interface on the switching chip usually has limited bandwidth since the interface is designed for carrying control/management messages. ServerSwitch overcomes this limitation by introducing additional NIC chips for a high bandwidth, yet low latency channel between the switching chip and the server; (2) ServerSwitch tries to provide a common set of APIs to program the switch chip. The APIs are designed to be as universal as possible. Ideally, the API is the same no matter what kind of switching chip is used.

Ripcord [9] mainly focuses on the DCN control plane. It currently uses OpenFlow switches as its data plane. Our work is orthogonal to their work. We envision that they can also use ServerSwitch to support new DCN such as BCube, and to support more routing schemes such as source routing and tag-based routing.

9 Conclusion

We have presented the design and implementation of ServerSwitch, a programmable and high performance platform for data center networks. ServerSwitch explores the design space of integrating a high performance, limited programmable ASIC switching chip with a powerful, fully programmable multicore commodity server.

ServerSwitch achieves easy-to-use programmability by using the server system to program and control the switching chip. The switching chip can be programmed to support a flexible packet header format and various user defined packet forwarding designs with line-rate without the server CPU intervening. By leveraging the low latency PCI-E interface and efficient server software design, we can implement software defined signaling and congestion control in the server CPU with low CPU overhead. The rich programmability provided by Server-

Switch can further enable new DCN services that need in-network data processing such as in-network caching.

We have built a ServerSwitch card and a whole ServerSwitch software stack. Our implementation experiences demonstrate that ServerSwitch can be fully constructed from commodity, inexpensive components. Our development experiences further show that ServerSwitch is easy to program, using the standard C/C++ language and development tool chains. We have used our ServerSwitch platform to construct several recently proposed DCN designs, including new DCN architectures BCube and PortLand, congestion control algorithm QCN, and DCN in-network caching service.

Our software API currently focuses on lookup table programmability and queue information query. Current switching chips also provide advanced features such as queue and buffer management, access control, and priority and fair queueing scheduling. We plan to extend our API to cover these features in our future work. We also plan to upgrade the current 1GE hardware to 10G in the next version. We expect that ServerSwitch may be used for networking research beyond DCN (*e.g.*, enterprise networking). We plan to release both the ServerSwitch card and the software package to the networking research community in the future.

Acknowledgements

We thank our shepherd Sylvia Ratnasamy and the anonymous NSDI reviewers for their valuable feedback on early versions of this paper. We thank Xiongfei Cai and Hao Zhang for their work on the initial ServerSwitch hardware design, the members of the Wireless and Networking Group and Zheng Zhang at Microsoft Research Asia for their support and feedback.

References

[1] `http://www.broadcom.com/collateral/pb/56330-PB01-R.pdf`.

[2] `http://www.suacommunity.com/SUA.aspx`.

[3] FM3000 Policy Engine, 2008. `http://www.fulcrummicro.com/documents/applications/FM3000_Policy_Engine.pdf`.

[4] ABU-LIBDEH, H., COSTA, P., ROWSTRON, A., O'SHEA, G., AND DONNELLY, A. Symbiotic Routing in Future Data Centers. In *ACM SIGCOMM* (2010).

[5] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *ACM SIGCOMM* (2008).

[6] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010).

[7] ALIZADEH, M., ATIKOGLU, B., KABBANI, A., LAKSHMIKANTHA, A., PAN, R., PRABHAKAR, B., AND SEAMAN, M. Data Center Transport Mechanisms: Congestion Control Theory and IEEE Standardization. In *46th Annual Allerton Conference on Communication, Control, and Computing,* (2008).

[8] CALDERON, M., SEDANO, M., AZCORRA, A., AND ALONSA, C. Active Network Support for Multicast Applications. *Network, IEEE 12*, 3 (may. 1998), 46 –52.

[9] CASADO, M., ET AL. Ripcord: a Module Platform for Data Center Networking. Tech. Rep. UCB/EECS-2010-93, Univeristy of California at Berkeley, 2010.

[10] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *ACM SOSP* (2009).

[11] GREENBERG, A., HAMILTON, J. R., JAIN, N., KANDULA, S., KIM, C., LAHIRI, P., MALTZ, D., PATEL, P., AND SENGUPTA, S. VL2: a Scalable and Flexible Data Center Network. In *ACM SIGCOMM* (2009).

[12] GREENHALGH, A., HUICI, F., HOERDT, M., PAPADIMITRIOU, P., HANDLEY, M., AND MATHY, L. Flow Processing and the Rise of Commodity Network Hardware. *SIGCOMM Comput. Commun. Rev. 39*, 2 (2009), 20–26.

[13] GUO, C., LU, G., LI, D., WU, H., ZHANG, X., SHI, Y., TIAN, C., ZHANG, Y., AND LU, S. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *ACM SIGCOMM* (2009).

[14] GUO, C., LU, G., WANG, H. J., YANG, S., KONG, C., SUN, P., WU, W., AND ZHANG, Y. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *ACM CoNext* (2010).

[15] GUO, C., WU, H., TAN, K., SHI, L., ZHANG, Y., AND LU, S. DCell: A Scalable and Fault Tolerant Network Structure for Data Centers. In *ACM SIGCOMM* (2008).

[16] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-Accelerated Software Router. In *ACM SIGCOMM* (2010).

[17] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Transactions on Computer Systems* (August 2000), 263–297.

[18] LEHMAN, L., GARLAND, S., AND TENNENHOUSE, D. Active Reliable Multicast. In *IEEE INFOCOM* (1998).

[19] LU, G., GUO, C., LI, Y., ZHOU, Z., YUAN, T., WU, H., XIONG, Y., GAO, R., AND ZHANG, Y. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. Tech. Rep. MSR-TR-2011-24, Microsoft Research, 2011.

[20] MILLER, D. J., WATTS, P. M., AND MOORE, A. W. Motivating Future Interconnects: A Differential Measurement Analysis of PCI Latency. In *ACM/IEEE ANCS* (2009).

[21] MOGUL, J. C., YALAGANDULA, P., TOURRILHES, J., MCGEER, R., BANERJEE, S., CONNORS, T., AND SHARMA, P. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *ACM HotNets-VII* (2008).

[22] MYSORE, R. N., ET AL. PortLand: a Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *ACM SIGCOMM* (2009).

[23] NAOUS, J., GIBB, G., BOLOUKI, S., AND MCKEOWN, N. NetFPGA: Reusable Router Architecture for Experimental Research. In *PRESTO* (2008).

[24] PAN, R. QCN Pseudo Code. `http://www.ieee802.org/1/files/public/docs2009/au-rong-qcn-serial-hai-v23.pdf`.

[25] SHAH, D., AND GUPTA, P. Fast Updating Algorithms for TCAMs. *IEEE Micro 21*, 1 (2001), 36–47.

[26] SHIEH, A., KANDULA, S., AND SIRER, E. G. Sidecar: Building Programmable Datacenter Networks without Programmable Switches. In *ACM HotNets* (2010).

TritonSort: A Balanced Large-Scale Sorting System

Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha†
Radhika Niranjan Mysore, Alexander Pucher*, Amin Vahdat
UC San Diego, UC Riverside†, and Vienna University of Technology*

Abstract—**We present TritonSort, a highly efficient, scalable sorting system. It is designed to process large datasets, and has been evaluated against as much as 100 TB of input data spread across 832 disks in 52 nodes at a rate of 0.916 TB/min. When evaluated against the annual Indy GraySort sorting benchmark, TritonSort is 60% better in absolute performance and has over six times the per-node efficiency of the previous record holder. In this paper, we describe the hardware and software architecture necessary to operate TritonSort at this level of efficiency. Through careful management of system resources to ensure cross-resource balance, we are able to sort data at approximately 80% of the disks' aggregate sequential write speed.**

We believe the work holds a number of lessons for balanced system design and for scale-out architectures in general. While many interesting systems are able to scale linearly with additional servers, per-server performance can lag behind per-server capacity by more than an order of magnitude. Bridging the gap between high scalability and high performance would enable either significantly cheaper systems that are able to do the same work or provide the ability to address significantly larger problem sets with the same infrastructure.

1 Introduction

The need for large-scale computing is increasing, driven by search engines, social networks, location-based services, and biological and scientific applications. The value of these applications is defined by the quality and quantity of data over which they operate, resulting in very high I/O and storage requirements. These Data-intensive Scalable Computing systems, or DISC systems[8], require searching and sorting large quantities of data spread across the network. Sorting forms the kernel of many data processing tasks in the datacenter, exercises computing, I/O, and storage resources, and is a key bottleneck for many large-scale systems.

Several new DISC software architectures have been developed recently, including MapReduce[9], the Google File System[11], Hadoop[22], and Dryad[14]. These systems are able to scale linearly with the number of nodes in the cluster, making it trivial to add new processing capability and storage capacity to an existing cluster by simply adding more nodes. This linear scalability is achieved in part by exposing parallel programming models to the user and by performing computation on data locally whenever possible. Hadoop clusters with thousands of nodes are now deployed in practice [23].

Despite this linear scaling behavior, per-node performance has lagged behind per-server capacity by more than an order of magnitude. A survey of several deployed DISC sorting systems[4] found that the impressive results obtained by operating at high scale mask a typically low individual per-node efficiency, requiring a larger-than-needed scale to meet application requirements. For example, among these systems as much as 94% of available disk I/O and 33% CPU capacity remained idle[4]. The largest known industrial Hadoop clusters achieve only 20 Mbps of average bandwidth for large-scale data sorting on machines theoretically capable of supporting a factor of 100 more throughput.

In this work we present TritonSort, a highly efficient sorting system designed to sort large volumes of data across dozens of nodes. We have applied it to data sets as large as 100 terabytes spread across 832 disks in 52 nodes. The key to TritonSort's efficiency is its *balanced* software architecture, which is able to effectively make use of a large amount of co-located storage per node, ensuring that the disks are kept as utilized as possible. Our results show the benefit of our design: evaluating TritonSort against the 'Indy' GraySort benchmark[19] resulted in a system that was able to sort 100TB of input tuples in about 60% of the absolute time of the previous record-holder, but with four times fewer resources, resulting in an increase in per-node efficiency by over a factor of six.

It is important to note that our focus in building TritonSort is to highlight the efficiency gains that can be obtained in building systems that process significant amounts of data through balancing computation, storage, memory, and network. Systems such as Hadoop and Dryad further support data-level replication, transparent node failure, and a generalized computational model, all of which are not currently present in TritonSort. However, in presenting TritonSort's hardware and software architecture, we describe several lessons learned in its construction that we believe are generalizable to other data processing systems. For example, our design relies

on a very high disk-to-node ratio as well as an explicit, application-level management of in-memory buffers to minimize disk seeks and thus increase read and write throughput. We choose buffer sizes to balance time spent processing multiple stages of our sort pipeline, and trade off the utilization of one resource for another.

Our experiences show that for a common datacenter workload, systems can be built with commodity hardware and open-source software that improve on per-node efficiency by an order of magnitude while still achieving scalability. Building such systems will either enable significantly cheaper systems to be able to do the same work or provide the ability to address significantly larger problem sets with the same infrastructure.

The primary contributions of this paper are: 1) the selection of a balanced hardware platform tuned to support a large-scale sort application, 2) a sort application implemented on top of a staged, pipeline-oriented software runtime that supports performance tuning via selection of appropriate buffer sizes and quantities, 3) an examination of projected sort performance when bottlenecks are removed, and 4) a discussion of the experience gained in building and deploying this prototype at scale.

2 Design Challenges

In this paper, we focus on designing systems that sort large datasets as an instance of the larger problem of building balanced systems. Here, we present our precise problem formulation, discuss the challenges involved, and outline the key insights underlying our approach.

2.1 Problem Formulation

We seek to design a system that sorts large volumes of input data. Based on the specification of the sort benchmark [19], our input data comprises 100 byte tuples with a 10 byte key and 90 byte value. We target deployments with input data on the order of tens to hundreds of TB of randomly-generated tuples. The input data is stored as a collection of files on persistent storage. The goal of a sorting system is to transform this input data into an ordered set of output files, also stored on persistent storage, such that the concatenation of these output files in order constitutes the sorted version of the input data. Our goal is to design and implement a sorting system that can sort datasets of the targeted size while achieving a favorable tradeoff between speed, resource utilization, and cost.

2.2 The Challenge of Efficient Sorting

Sorting large datasets places stress on several resources in a cluster. First, storing tens to hundreds of TB of input and output data demands a large amount of storage capacity. Given the size of the data and modern commodity hard drive capacities, the data must be stored across several storage devices and almost certainly across many machines. Second, reading the input data and writing the output data across many disks simultaneously places load on both storage devices and I/O controllers. Third, since the tuples are distributed randomly across the input files, almost all of the large dataset to be sorted will have to be sent over the network. Finally, comparing tuples in order to sort them requires a non-trivial amount of compute power. This combination of demands makes designing a sorting system that efficiently utilizes all of these resources challenging.

Our key design principle to ensure good resource utilization is to construct a balanced system—a system that drives all resources at as close to 100% utilization as possible. For any given application and workload, there will be an ideal configuration of hardware resources in keeping with the application's demands on these resources. In practice, the set of hardware configurations available is limited by the availability of components (one cannot currently, for example, buy a processor with exactly 13 cores), and so a configuration must be chosen that best meets the application's demands. Once that hardware configuration is determined, the application must be architected to suitably exploit the full capabilities of the deployed hardware. In the following section, we outline our considerations in designing such a balanced system, including our choice of a specific hardware and software architecture. We did not choose this platform with sorting in mind, and so we believe that our design generalizes to other DISC problems as well.

2.3 Design Considerations

Our system's design is motivated by three main considerations. First, we rely only on commodity hardware components. This is both to keep the costs of our system relatively low and to have our system be representative of today's data centers so that the lessons we learn can be applied to other applications with workload characteristics similar to those of sort. Hence, we do not make use of networking substrates such as Infiniband that provide high network bandwidth at high cost. Also, despite the recent emergence of solid state drives (SSDs) that provide higher I/O rates, we chose to use hard disks because they continue to provide the most affordable option for high capacity storage and streaming I/O. We believe that properly-architected sorting software should not stress random I/O behavior, where SSDs currently excel.

Second, we focus our software architecture on minimizing disk seeks. In the particular hardware configuration we chose, the key bottleneck for sort among the various system resources is disk I/O bandwidth. Hence, the primary goal of the system is to enable all disks to operate continuously at peak bandwidth. The main challenge in sustaining peak disk bandwidth is to minimize

the amount of time the disks spend seeking, since any time seeking is not spent transferring data.

Third, we choose to focus on hardware architectures whose total memory cannot contain the entire dataset. One possible implementation of sort is to read all the input data into memory, appropriately shuffle the data across machines in the cluster, sort the local in-memory data on each machine, and then write the sorted data to the local disks. Note that in this case, every tuple is read from and written to persistent storage exactly once. However, this implementation would require an amount of memory at least equal to the amount of input data; given that the cost per GB of RAM is over 70 times more than that of disks, such a design would significantly drive up costs and be infeasible for large input datasets.

Instead, we pursue an alternative implementation wherein every tuple is read and written multiple times from disk before the data is completely sorted. Storing intermediate results on disk makes the system's memory requirement far more modest. Sorting data on clusters that have less memory than the total amount of data to be sorted requires every input tuple to be read and written at least twice [1]. Since every additional read and write increases the time to sort, we seek to achieve exactly this lower bound to maximize system performance.

2.4 Hardware Architecture

To determine the right hardware configuration for our application, we make the following observations about the sort workload. First, the application needs to read every byte of the input data and the size of the input is equal to that of the output. Since the "working set" is so large, it does not make sense to separate the cluster into computation-heavy and storage-heavy regions. Instead, we provision each server in the cluster with an equal amount of processing power and disks.

Second, almost all of the data needs to be exchanged between machines since input data is randomly distributed throughout the cluster and adjacent tuples in the sorted sequence must reside on the same machine. To balance the system, we need to ensure that this all-to-all shuffling of data can happen in parallel without network bandwidth becoming a bottleneck. Since we focus on using commodity components, we use an Ethernet network fabric. Commodity Ethernet is available in a set of discrete bandwidth levels—1 Gbps, 10 Gbps, and 40 Gbps—with cost increasing proportional to throughput (see Table 1). Given our choice of 7.2k-RPM disks for storage, a 1 Gbps network can accommodate at most one disk per server without the network throttling disk I/O. Therefore, we settle on a 10 Gbps network; 40 Gbps Ethernet has yet to mature and hence is still cost prohibitive. To balance a 10 Gbps network with disk I/O, we use a server that can host 16 disks. Based on the options available commercially for such a server, we use a server that hosts 16 disks and 8 CPU cores. The choice of 8 cores was driven by the available processor packaging: two physical quad-core CPUs. The larger the number of separate threads, the more stages that can be isolated from each other. In our experience, the actual speed of each of these cores was a secondary consideration.

Storage			
Type	Capacity	R/W throughput	Price
7.2k-RPM	500 GB	90-100 MBps	$200
15k-RPM	150 GB	150 MBps	$290
SSD	64 GB	250 MBps	$450

Network	
Type	Cost/port
1 Gbps Ethernet	$33
10 Gbps Ethernet	$480

Server	
Type	Cost
8 disks, 8 CPU cores	$5,050
8 disks, 16 CPU cores	$5,450
16 disks, 16 CPU cores	$7,550

Table 1: Resource options considered for constructing a cluster for a balanced sorting system. These values are estimates as of January, 2010.

Third, sort demands both significant capacity and I/O requirements from storage since tens to hundreds of TB of data is to be stored and all the data is to be read and written twice. To determine the best storage option given these requirements, we survey a range of hard disk options shown in Table 1. We find that 7.2k-RPM SATA disks provide the most cost-effective option in terms of balancing $ per GB and $ per read/write MBps (assuming we can achieve streaming I/O). To allow 16 disks to operate at full streaming I/O throughput, we require storage controllers that are able to sustain at least 1600 MBps of streaming bandwidth. Because of the PCI bus' bandwidth limitations, our hardware design necessitated two 8x PCI drive controllers, each supporting 8 disks.

The final design choice in provisioning our cluster is the amount of memory each server should have. The primary purpose of memory in our system is to enable large amounts of data buffering so that we can read from and write to the disk in large chunks. The larger these chunks become, the more data can be read or written before seeking is required. We initially provisioned each of our machines with 12 GB of memory; however, during development we realized that 24 GB was required to provide sufficiently large writes, and so the machines were upgraded. We discuss this addition when we present our

architecture in Section 3. One of the key takeaways from our work is the important role that buffering plays in enabling high utilization of the network, disk, and CPU. Determining the appropriate amount of memory buffering is not straightforward and we leave to future work techniques that help automate this process.

2.5 Software Architecture

To maximize cluster resource utilization, we need to design an appropriate software architecture. There are a range of possible software architectures in keeping with our constraint of reading and writing every input tuple at most twice. The class of architectures upon which we focus share a similar basic structure. These architectures consist of two phases separated by a distributed barrier, so that all nodes must complete phase one before phase two begins. In the first phase, input data is read from disk and routed to the node upon which it will ultimately reside. Each node is responsible for storing a disjoint portion of the key space. When data arrives at its destination node, that node writes the data to its local disks. In the second phase, each node sorts the data on its local disks in parallel. At the end of the second phase, each node has a portion of the final sorted sequence stored on its local disks, and the sorted sequences stored on all nodes can be concatenated together to form the final sorted sequence.

There are several possible implementations of this general architecture, but any implementation contains at least a few basic software elements. These software elements include *Readers* that read data from on-disk files into in-memory buffers, *Writers* that write buffers to disk, *Distributors* that distribute a buffer's tuples across a set of logical divisions and *Sorters* that sort buffers.

Our initial implementation of TritonSort was designed as a distributed parallel external merge-sort. This architecture, which we will call the Heaper-Merger architecture, is structured as follows. In phase one, Readers read from the input files into buffers, which are sorted by Sorters. Each sorted buffer is then passed to a Distributor, which splits the buffer into a sorted chunk per node and sends each chunk to its corresponding node. Once received, these sorted chunks are heap-sorted by software elements called Heapers in batches and each resulting sorted batch is written to an intermediate file on disk. In the second phase, software elements called Mergers merge-sort the intermediate files on a given disk into a single sorted output file.

The problem with the Heaper-Merger architecture is that it does not scale well. In order to prevent the Heaper in phase one from becoming a bottleneck, the length of the sorted runs that the Heaper generates is usually fairly small, on the order of a few hundred megabytes. As a consequence, the number of intermediate files that the Merger must merge in phase two grows quickly as the

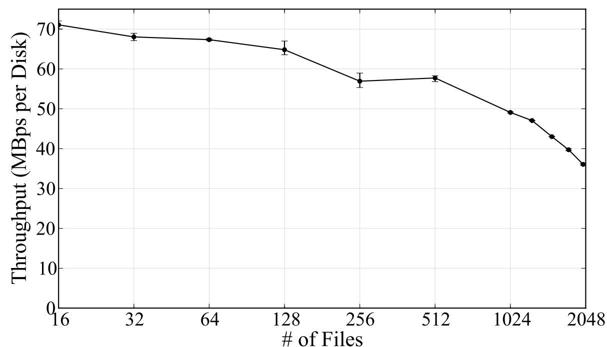


Figure 1: Performance of a Heaper-Merger sort implementation in microbenchmark on a 200GB per disk parallel external merge-sort as a function of the number of files merged per disk.

size of the input data increases. This reduces the amount of data from each intermediate file that can be buffered at a time by the Merger and requires that the merger fetch additional data from files much more frequently, causing many additional seeks.

To demonstrate this problem, we implemented a simple Heaper-Merger sort module in microbenchmark. We chose to sort 200GB per disk in parallel across all the disks to simulate the system's performance during a 100TB sort. Each disk's 200GB data set is partitioned among an increasingly large number of files. Each node's memory is divided such that each input file and each output file can be double-buffered. As shown in Figure 1, increasing the number of files being merged causes throughput to decrease dramatically as the number of files increases above 1000.

TritonSort uses an alternative architecture with similar software elements as above and again involving two phases. We partition the input data into a set of logical partitions; with D physical disks and L logical partitions, each logical partition corresponds to a contiguous $\frac{1}{L}^{th}$ fraction of the key space and each physical disk hosts $\frac{L}{D}$ logical partitions. In the first phase, Readers pass buffers directly to Distributors. A Distributor maps the key of every tuple in its input buffer to its corresponding logical partition and sends that tuple over the network to the machine that hosts this logical partition. Tuples for a given logical partition are buffered in memory and written to disk in large chunks in order to seek as little as possible. In the second phase, each logical partition is read into an in-memory buffer, that buffer is sorted, and the sorted buffer is written to disk. This scheme bypasses the seek limits of the earlier mergesort-based approach. Also, by appropriately choosing the value of L, we can ensure that logical partitions can be read, sorted and written in parallel in the second phase. Since our testbed nodes have 24GB of RAM, to ensure this condition we set the num-

ber of logical partitions per node to 2520 so that each logical partition contains less than 1GB of tuples when we sort 100 TB on 52 nodes. We explain this architecture in more detail in the context of our implementation in the next section.

3 Design and Implementation

TritonSort is a distributed, staged, pipeline-oriented dataflow processing system. In this section, we describe TritonSort's design and motivate our design decisions for each stage in its processing pipeline.

3.1 Architecture Overview

Figures 2 and 7 show the stages of a TritonSort program. Stages in TritonSort are organized in a directed graph (with cycles permitted). Each stage in TritonSort implements part of the data processing pipeline and either sources, sinks, or transmutes data flowing through it.

Each stage is implemented by two types of logical entities—several *workers* and a single *WorkerTracker* . Each worker runs in its own thread and maintains its own local queue of pending work. We refer to the discrete pieces of data over which workers operate as *work units* or simply as *work*. The WorkerTracker is responsible for accepting work for its stage and assigning that work to workers by enqueueing the work onto the worker's work queue. In each phase, all the workers for all stages in that phase run in parallel.

Upon starting up, a worker initializes any required internal state and then waits for work. When work arrives, the worker executes a stage-specific *run()* method that implements the specific function of the stage, handling work in one of three ways. First, it can accept an individual work unit, execute the *run()* method over it, and then wait for new work. Second, it can accept a batch of work (up to a configurable size) that has been enqueued by the WorkerTracker for its stage. Lastly, it can keep its *run()* method active, polling for new work explicitly. TritonSort stages implement each of these methods, as described below. In the process of running, a stage can produce work for a downstream stage and optionally specify the worker to which that work should be directed. If a worker does not specify a destination worker, work units are assigned to workers round-robin.

In the process of executing its *run()* method, a worker can get buffers from and return buffers to a shared pool of buffers. This buffer pool can be shared among the workers of a single stage, but is typically shared between workers in pairs of stages with the upstream stage getting buffers from the pool and the downstream stage putting them back. When getting a buffer from a pool, a stage can specify whether or not it wants to block waiting for a buffer to become available if the pool is empty.

3.2 Sort Architecture

We implement sort in two phases. First, we perform distribution sort to partition the input data across L logical partitions evenly distributed across all nodes in the cluster. Each logical partition is stored in its own *logical disk*. All logical disks are of identical maximum size $size_{LD}$ and consist of files on the local file system.

The value of $size_{LD}$ is chosen such that logical disks from each physical disk can be read, sorted and written in parallel in the second phase, ensuring maximum resource utilization. Therefore, if the size of the input data is $size_{input}$, there are $L = \frac{size_{input}}{size_{LD}}$ logical disks in the system. In phase two, the tuples in each logical disk get sorted locally and written to an output file. This implementation satisfies our design goal of reading and writing each tuple twice.

To determine which logical disk holds which tuples, we logically partition the 10-byte key space into L even divisions. We logically order the logical disks such that the kth logical disk holds tuples in the kth division. Sorting each logical disk produces a collection of output files, each of which contains sorted tuples in a given partition. Hence, the ordered collection of output files represents the sorted version of the data. In this paper, we assume that tuples' keys are distributed uniformly over the key range which ensures that each logical disk is approximately the same size; we discuss how TritonSort can be made to handle non-uniform key ranges in Section 6.1.

To ensure that we can utilize as much read/write bandwidth as possible on each disk, we partition the disks on each node into two groups of 8 disks each. One group of disks holds input and output files; we refer to these disks as the input disks in phase one and as the output disks in phase two. The other group holds intermediate files; we refer to these disks as the intermediate disks. In phase one, input files are read from the input disks and intermediate files are written to the intermediate disks. In phase two, intermediate files are read from the intermediate disks and output files are written to the output disks. Thus, the same disk is never concurrently read from and written to, which prevents unnecessary seeking.

3.3 TritonSort Architecture: Phase One

Phase one of TritonSort, diagrammed in Figure 2, is responsible for reading input tuples off of the input disks, distributing those tuples over to the network to the nodes on which they belong, and storing them into the logical disks in which they belong.

Reader: Each Reader is assigned an input disk and is responsible for reading input data off of that disk. It does this by filling 80 MB ProducerBuffers with input data. We chose this size because it is large enough to obtain near sequential throughput from the disk.

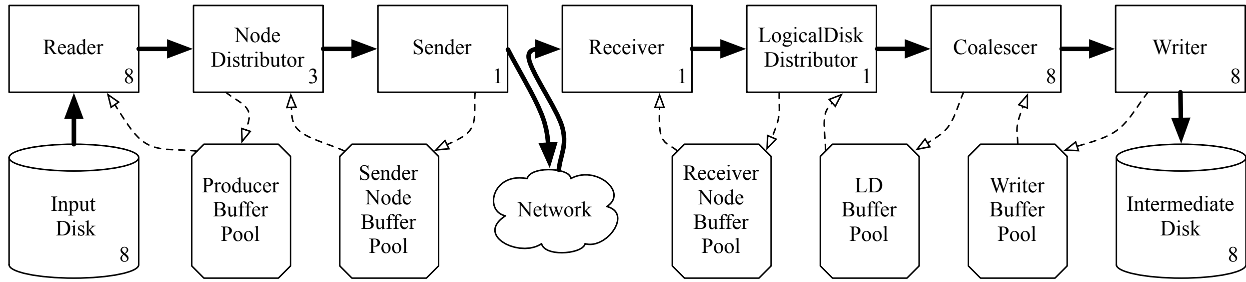


Figure 2: Block diagram of TritonSort's phase one architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

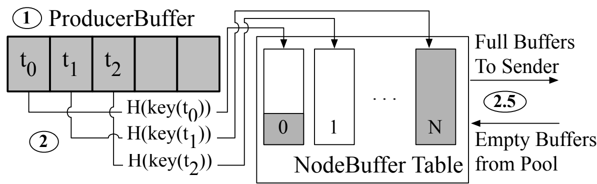


Figure 3: The NodeDistributor stage, responsible for partitioning tuples by destination node.

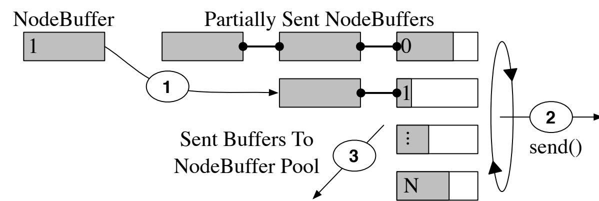


Figure 4: The Sender stage, responsible for sending data to other nodes.

NodeDistributor: A NodeDistributor (shown in Figure 3) receives a ProducerBuffer from a Reader and is responsible for partitioning the tuples in that buffer across the machines in the cluster. It maintains an internal data structure called a *NodeBuffer table*, which is an array of NodeBuffers, one for each of the nodes in the cluster. A NodeBuffer contains tuples belonging to the same destination machine. Its size was chosen to be the size of the ProducerBuffer divided by the number of nodes, and is approximately 1.6 MB in size for the scales we consider in this paper.

The NodeDistributor scans the ProducerBuffer tuple by tuple. For each tuple, it computes a hash function $H(k)$ over the tuple's key k that maps the tuple to a unique host in the range $[0, N-1]$. It uses the NodeBuffer table to select a NodeBuffer corresponding to host $H(k)$ and appends the tuple to the end of that buffer. If that append operation causes the buffer to become full, the NodeDistributor removes the NodeBuffer from the NodeBuffer table and sends it downstream to the Sender stage. It then gets a new NodeBuffer from the NodeBuffer pool and inserts that buffer into the newly empty slot in the NodeBuffer table. Once the NodeDistributor is finished processing a ProducerBuffer, it returns that buffer back to the ProducerBuffer pool.

Sender: The Sender stage (shown in Figure 4) is responsible for taking NodeBuffers from the upstream NodeDistributor stage and transmitting them over the network to each of the other nodes in the cluster. Each Sender maintains a separate TCP socket per peer node in the cluster. The Sender stage can be implemented in a multi-threaded or a single-threaded manner. In the multi-threaded case, N Sender workers are instantiated in their own threads, one for each destination node. Each Sender worker simply issues a blocking *send()* call on each NodeBuffer it receives from the upstream NodeDistributor stage, sending tuples in the buffer to the appropriate destination node over the socket open to that node. When all the tuples in a buffer have been sent, the NodeBuffer is returned to its pool, and the next one is processed. For reasons described in Section 4.1, we choose a single-threaded Sender implementation instead. Here, the Sender interleaves the sending of data across all the destination nodes in small non-blocking chunks, so as to avoid the overhead of having to activate and deactivate individual threads for each send operation to each peer.

Unlike most other stages, which process a single unit of work during each invocation of their *run()* method, the Sender continuously processes NodeBuffers as it runs, receiving new work as it becomes available from the NodeDistributor stage. This is because the Sender must remain active to alternate between two tasks: accepting incoming NodeBuffers from upstage NodeDistributors, and sending data from accepted NodeBuffers downstream. To facilitate accepting incoming NodeBuffers, each Sender maintains a set of NodeBuffer lists, one for each destination host. Initially these lists are empty. The Sender appends each NodeBuffer it receives onto the list of NodeBuffers corresponding to the incoming NodeBuffer's destination node.

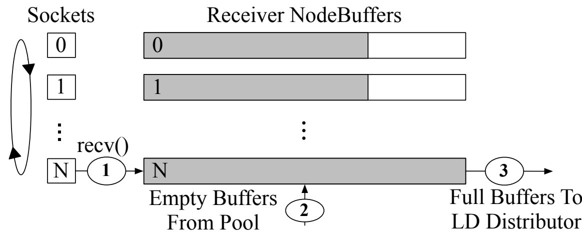


Figure 5: The Receiver stage, responsible for receiving data from other nodes' Sender stages.

To send data across the network, the Sender loops through the elements in the set of NodeBuffer lists. If the list is non-empty, the Sender accesses the NodeBuffer at the head of the list, and sends a fixed-sized amount of data to the appropriate destination host using a non-blocking *send()* call. If the call succeeds and some amount of data was sent, then the NodeBuffer at the head of the list is updated to note the amount of its contents that have been successfully sent so far. If the *send()* call fails, because the TCP send buffer for that socket is full, that buffer is simply skipped and the Sender moves on to the next destination host. When all of the data from a particular NodeBuffer is successfully sent, the Sender returns that buffer back to its pool.

Receiver: The Receiver stage, shown in Figure 5, is responsible for receiving data from other nodes in the cluster, appending that data onto a set of NodeBuffers, and passing those NodeBuffers downstream to the LogicalDiskDistributor stage. In TritonSort, the Receiver stage is instantiated with a single worker. On starting up, the Receiver opens a server socket and accepts incoming connections from Sender workers on remote nodes. Its *run()* method begins by getting a set of NodeBuffers from a pool of such buffers, one for each source node. The Receiver then loops through each of the open sockets, reading up to 16KB of data at a time into the NodeBuffer for that source node using a non-blocking *recv()* call. This small socket read size is due to the rate-limiting fix that we explain in Section 4.1. If data is returned by that call, it is appended to the end of the NodeBuffer. If the append would exceed the size of the NodeBuffer, that buffer is sent downstream to the LogicalDiskDistributor stage, and a new NodeBuffer is retrieved from the pool to replace the NodeBuffer that was sent.

LogicalDiskDistributor: The LogicalDiskDistributor stage, shown in Figure 6, receives NodeBuffers from the Receiver that contain tuples destined for logical disks on its node. LogicalDiskDistributors are responsible for distributing tuples to appropriate logical disks and sending groups of tuples destined for the same logical disk to the downstream Writer stage.

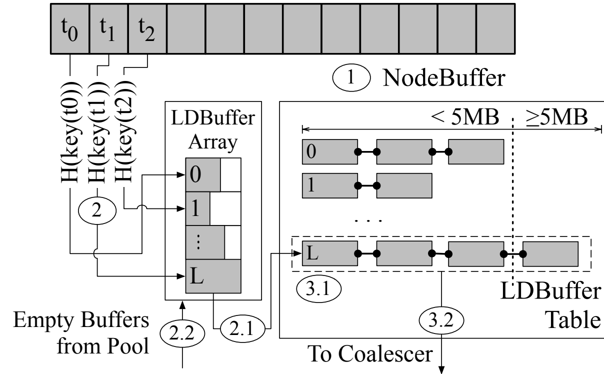


Figure 6: The LogicalDiskDistributor stage, responsible for distributing tuples across logical disks and buffering sufficient data to allow for large writes.

The LogicalDiskDistributor's design is driven by the need to buffer enough data to issue large writes and thereby minimize disk seeks and achieve high bandwidth. Internal to the LogicalDiskDistributor are two data structures: an array of LDBuffers, one per logical disk, and an LDBufferTable. An LDBuffer is a buffer of tuples destined to the same logical disk. Each LDBuffer is 12,800 bytes long, which is the least common multiple of the tuple size (100 bytes) and the direct I/O write size (512 bytes). The LDBufferTable is an array of LDBuffer lists, one list per logical disk. Additionally, LogicalDiskDistributor maintains a pool of LDBuffers, containing 1.25 million LDBuffers, accounting for 20 of each machine's 24 GB of memory.

Algorithm 1 The LogicalDiskDistributor stage

```
 1: NodeBuffer ← getNewWork()
 2: {Drain NodeBuffer into the LDBufferArray}
 3: for all tuples t in NodeBuffer do
 4:     dst = H(key(t))
 5:     LDBufferArray[dst].append(t)
 6:     if LDBufferArray[dst].isFull() then
 7:         LDTable.insert(LDBufferArray[dst])
 8:         LDBufferArray[dst] = getEmptyLDBuffer()
 9:     end if
10: end for
11: {Send full LDBufferLists to the Coalescer}
12: for all physical disks d do
13:     while LDTable.sizeOfLongestList(d) ≥ 5MB do
14:         ld ← LDTable.getLongestList(d)
15:         Coalescer.pushNewWork(ld)
16:     end while
17: end for
```

The operation of a LogicalDiskDistributor worker is described in Algorithm 1. In Line 1, a full NodeBuffer is pushed to the LogicalDiskDistributor by the Receiver.

Lines 3-10 are responsible for draining that NodeBuffer tuple by tuple into an array of LDBuffers, indexed by the logical disk to which the tuple belongs. Lines 12-17 examine the LDBufferTable, looking for logical disk lists that have accumulated enough data to write out to disk. We buffer at least 5 MB of data for each logical disk before flushing that data to disk to prevent many small write requests from being issued if the pipeline temporarily stalls. When the minimum threshold of 5 MB is met for any particular physical disk, the longest LDBuffer list for that disk is passed to the Coalescer stage on Line 15.

The original design of the LogicalDiskDistributor only used the LDBuffer array described above and used much larger LDBuffers (~10MB each) rather than many small LDBuffers. The Coalescer stage (described below) did not exist; instead, the LogicalDiskDistributor transferred the larger LDBuffers directly to the Writer stage.

This design was abandoned due to its inefficient use of memory. Temporary imbalances in input distribution could cause LDBuffers for different logical disks to fill at different rates. This, in turn, could cause an LDBuffer to become full when many other LDBuffers in the array are only partially full. If an LDBuffer is not available to replace the full buffer, the system must block (either immediately or when an input tuple is destined for that buffer's logical disk) until an LDBuffer becomes available. One obvious solution to this problem is to allow partially full LDBuffers to be sent to the Writers at the cost of lower Writer throughput. This scheme introduced the further problem that the unused portions of the LDBuffers waiting to be written could not be used by the LogicalDiskDistributor. In an effort to reduce the amount of memory wasted in this way, we migrated to the current architecture, which allows small LDBuffers to be dynamically reallocated to different logical disks as the need arises. This comes at the cost of additional computational overhead and memory copies, but we deem this cost to be acceptable due to the small cost of memory copies relative to disk seeks.

Coalescer: The operation of the Coalescer stage is simple. A Coalescer will copy tuples from each LDBuffer in its input LDBuffer list into a WriterBuffer and pass that WriterBuffer to the Writer stage. It then returns the LDBuffers in the list to the LDBuffer pool.

Originally, the LogicalDiskDistributor stage did the work of the Coalescer stage. While optimizing the system, however, we realized that the non-trivial amount of time spent merging LDBuffers into a single WriterBuffer could be better spent processing additional NodeBuffers.

Writer: The operation of the Writer stage is also quite simple. When a Coalescer pushes a WriterBuffer to it, the Writer worker will determine the logical disk corresponding to that WriterBuffer and write out the data using a blocking *write()* system call. When the write completes, the WriterBuffer is returned to the pool.

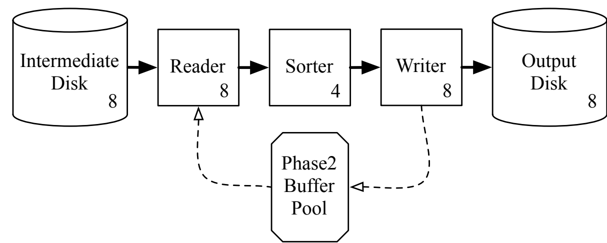


Figure 7: Block diagram of TritonSort's phase two architecture. The number of workers for a stage is indicated in the lower-right corner of that stage's block, and the number of disks of each type is indicated in the lower-right corner of that disk's block.

3.4 TritonSort Architecture: Phase Two

Once phase one completes, all of the tuples from the input dataset are stored in appropriate logical disks across the cluster's intermediate disks. In phase two, each of these unsorted logical disks is read into memory, sorted, and written out to an output disk. The pipeline is straightforward: Reader and Writer workers issue sequential, streaming I/O requests to the appropriate disk, and Sorter workers operate entirely in memory.

Reader: The phase two Reader stage is identical to the phase one Reader stage, except that it reads into a PhaseTwoBuffer, which is the size of a logical disk.

Sorter: The Sorter stage performs an in-memory sort on a PhaseTwoBuffer. A variety of sort algorithms can be used to implement this stage, however we selected the use of radix sort due to its speed. Radix sort requires additional memory overhead compared to an in-place sort like QuickSort, and so the sizes of our logical disks have to be sized appropriately so that enough Reader–Sorter–Writer pipelines can operate in parallel. Our version of radix sort first scans the buffer, constructing a set of structures containing a pointer to each tuple's key and a pointer to the tuple itself. These structures are then sorted by key. Once the structures have been sorted, they are used to rearrange the tuples in the buffer in-place. This reduces the memory overhead for each Sorter substantially at the cost of additional memory copies.

Writer: The phase two Writer writes a PhaseTwoBuffer sequentially to a file on an output disk. As in phase one, each Writer is responsible for writes to a single output disk.

Because the phase two pipeline operates at the granularity of a logical disk, we can operate several of these pipelines in parallel, limited by either the number of cores in each system (we can't have more pipelines than cores without sacrificing performance because the Sorter is CPU-bound), the amount of memory in the system

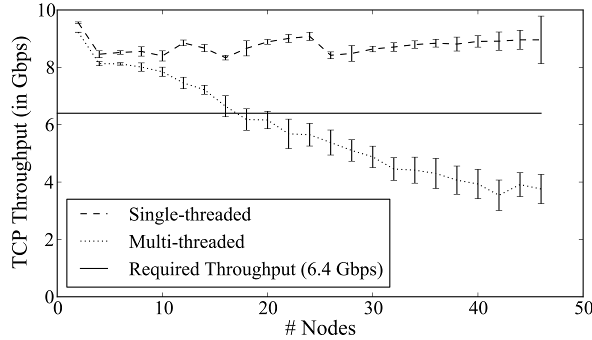


Figure 9: Comparing the scalability of single-threaded and multi-threaded Receiver implementations

(each pipeline requires at least three times the size of a logical disk to be able to read, sort, and write in parallel), or the throughput of the disks. In our case, the limiting factor is the output disk bandwidth. To host one phase two pipeline per input disk requires storing 24 logical disks in memory at a time. To accomplish this, we set $size_{LD}$ to $850MB$, using most of the 24 GB of RAM available on each node and allowing for additional memory required by the operating system. To sort $850MB$ logical disks fast enough to not block the Reader and Writer stages, we find that four Sorters suffice.

3.5 Stage and Buffer Sizing

One of the major requirements for operating TritonSort at near disk speed is ensuring cross-stage balance. Each stage has an intrinsic execution time, either based on the speed of the device to which it interfaces (e.g., disks or network links), or based on the amount of CPU time it requires to process a work unit. Figure 8 shows the speed and performance of each stage in the pipeline. In our implementation, we are limited by the speed of the Writer stage in both phases one and two.

4 Optimizations

In implementing the TritonSort architecture, we learned that several non-obvious optimizations were necessary to meet our desired goal of driving every disk at full utilization. Here, we present the key takeaways from our experience. In each case, we believe these lessons generalize to a wide variety of DISC systems.

4.1 Network

For TritonSort to operate at the aggregate sequential streaming bandwidth of all of its disks, the network must be able to sustain the read throughput of eight disks while data is being shuffled among nodes in the first phase. Since the 7.2k-RPM disks we use deliver at most 100 MBps of sequential read throughput (Table 1), the network must be able to sustain 6.4 Gbps of all-pairs bandwidth, irrespective of the number of nodes in the cluster.

It is well-known that sustaining high-bandwidth flows in datacenter networks, especially all-to-all patterns, is a significant challenge. Reasons for this include commodity datacenter network hardware, incast, queue buildup, and buffer pressure[2]. Since we could not employ a strategy like that presented in [2] to provide fair but high bandwidth flow rates among the senders, we chose instead to artificially rate limit each flow at the Sender stage to its calculated fair share by forcing the sockets to be receive window limited. This works for TritonSort because 1) each machine sends and receives at approximately the same rate, 2) all the nodes share the same RTT since they are interconnected by a single switch, and 3) our switch does not impose an oversubscription factor. In this case, each Sender should ideally send at a rate of $(6.4/N)$ Gbps, or 123 Mbps with a cluster of 52 nodes. Given that our network offers approximately $100\mu sec$ RTTs, a receiver window size of $8 - 16$ KB ensures that the flows will not impose queue buildup or buffer pressure on other flows.

Initially, we chose a straightforward multi-threaded design for the Sender and Receiver stages in which there were N Senders and N Receivers, one for each TritonSort node. In this design, each Sender issues blocking *send()* calls on a NodeBuffer until it is sent. Likewise, on the destination node, each Receiver repeatedly issues blocking *recv()* calls until a NodeBuffer has been received. Because the number of CPU hyperthreads on each of our nodes is typically much smaller than $2N$, we pinned all Senders' threads to a single hyperthread and all Receivers' threads to a single separate hyperthread.

Figure 9 shows that this multi-threaded approach does not scale well with the number of nodes, dropping below 4 Gbps at scale. This poor performance is due to thread scheduling overheads at the end hosts. 16 KB TCP receive buffers fill up much faster than connections that are not window-limited. At the rate of 123 MBps, a 16 KB buffer will fill up in just over 1 ms, causing the Sender to stop sending. Thus, the Receiver stage must clear out each of its buffers at that rate. Since there are 52 such buffers, a Receiver must visit and clear a receive buffer in just over 20 μs. A Receiver worker thread cannot drain the socket, block, go to sleep, and get woken up again fast enough to service buffers at this rate.

To circumvent this problem we implemented a single-threaded, non-blocking receiver that scans through each socket in round-robin order, copying out any available data and storing it in a NodeBuffer during each pass through the array of open sockets. This implementation is able to clear each socket's receiver buffer faster than the arrival rate of incoming data. Figure 9 shows that this design scales well as the cluster grows.

Worker Type	Size Of Input (MB)	Runtime (ms)	# Workers	Throughput (in MBps)	Total Throughput (in MBps)
Reader	81.92	958.48	8	85	683
NodeDistributor	81.92	263.54	3	310	932
LogicalDiskDistributor	1.65	2.42	1	683	683
Coalescer	10.60	4.56	8	2,324	18,593
Writer	10.60	141.07	8	75	601
Phase two Reader	762.95	8,238	8	92	740
Phase two Sorter	762.95	2,802	4	272	1089
Phase two Writer	762.95	8,512	8	89	717

Figure 8: Median stage runtimes for a 52-node, 100TB sort, excluding the amount of time spent waiting for buffers.

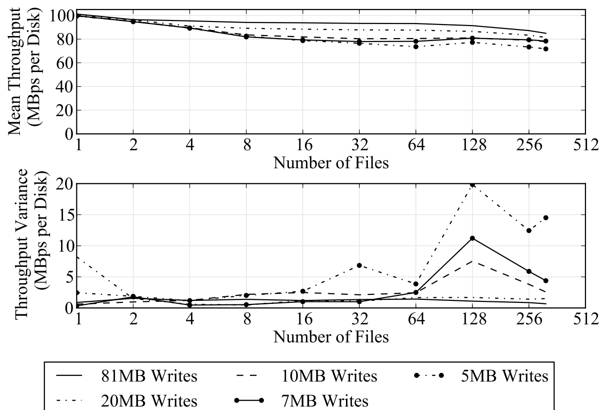


Figure 10: Microbenchmark indicating the ideal disk throughput as a function of write size

4.2 Minimizing Disk Seeks

Key to making the TritonSort pipeline efficient is minimizing the total amount of time spent performing disk seeks, both while writing data in phase one, and while reading that data in phase two. As individual write sizes get smaller, the throughput drops, since the disk must occasionally seek between individual write operations. Figure 10 shows disk write throughput measured by a synthetic workload generator writing to a configurable set of files with different write sizes. Ideally, the Writer would receive WriterBuffers large enough that it can write them out at close to the sequential rate of the disk, e.g., 80 MB. However, the amount of available memory limits TritonSort's write sizes. Since the tuple space is uniformly distributed across the logical disks, the LogicalDiskDistributor will fill its LDBuffers at approximately a uniform rate. Buffering 80 MB worth of tuples for a given logical disk before writing to disk would cause the buffers associated with all of the other logical disks to become approximately as full. This would mandate significantly higher memory needs than what is available in our hardware architecture. Hence, the LogicalDiskDistributor stage must emit smaller WriterBuffers, and it must interleave writes to different logical disks.

4.3 The Importance of File Layout

The physical layout of individual logical disk files plays a strong role in trading off performance between the phase one Writer and the phase two Reader. One strategy is to append to the logical disk files in a log-structured manner, in which a WriterBuffer for one logical disk is immediately appended after the WriterBuffer for a different logical disk. This is possible if the logical disks' blocks are allocated on demand. It has the advantage of making the phase one Writer highly performant, since it minimizes seeks and leads to near-sequential write performance. On the other hand, when a phase two Reader begins reading a particular logical disk, the underlying physical disk will need to seek frequently to read each of the WriterBuffers making up the logical disk.

An alternative approach is to greedily allocate all of the blocks for each of the logical disks at start time, ensuring that all of a logical disk's blocks are physically contiguous on the underlying disk. This can be accomplished with the *fallocate()* system call, which provides a hint to the file system to pre-allocate blocks. In this scheme, interleaved writes of WriterBuffers for different logical disks will require seeking since two subsequent writes to different logical disks will need to write to different contiguous regions on the disk. However, in phase two, the Reader will be able to sequentially read an entire logical disk with minimal seeking. We also use fallocate() on input and output files so that phase one Readers and phase two Writers seek as little as possible.

The location of output files on the output disks also has a dramatic effect on phase two's performance. If we do not delete the input files before starting phase two, the output files are allocated space on the interior cylinders of the disk. When evaluating phase two's performance on a 100 TB sort, we found that we could write to the interior cylinders of the disk at an average rate of 64 MBps. When we deleted the input files before phase two began, ensuring that the output files would be written to the exterior cylinders of the disk, this rate jumped to 84 MBps.

For the evaluations in Section 5, we delete the input files before starting phase two. For reference, the fastest we have been able to write to the disks in microbenchmark has been approximately 90 MBps.

4.4 CPU Scheduling

Modern operating systems support a wide variety of static and dynamic CPU scheduling approaches, and there has been considerable research into scheduling disciplines for data processing systems. We put a significant amount of effort into isolating stages from one another by setting the processor affinities of worker threads explicitly, but we eventually discovered that using the default Linux scheduler results in a steady-state performance that is only about 5% worse than any custom scheduling policy we devised. In our evaluation, we use our custom scheduling policy unless otherwise specified.

4.5 Pipeline Demand Feedback

Initially, TritonSort was entirely "push"-based, meaning that a worker only processed work when it was pushed to it from a preceding stage. While simple to design, certain stages perform sub-optimally when they are unable to send feedback back in the pipeline as to what work they are capable of doing. For example, the throughput of the Writer stage in phase one is limited by the latency of writes to the intermediate disks, which is governed by the sizes of WriterBuffers sent to it as well as the physical layout of logical disks (due to the effects of seek and rotational delay). In its naïve implementation, the LogicalDiskDistributor sends work to the Writer stage based on which of its LDBuffer lists is longest with no regard to how lightly or heavily loaded the Writers themselves are. This can result in an imbalance of work across Writers, with some Writers idle and others struggling to process a long queue of work. This imbalance can destabilize the whole pipeline and lower total throughput.

To address this problem, we must effectively communicate information about the sizes of Writers' work queues to upstream stages. We do this by creating a pool of *write tokens*. Every write token is assigned a single "parent" Writer. We assign parent Writers in round-robin order to tokens as the tokens are created and create a number of tokens equal to the number of WriterBuffers. When the LogicalDiskDistributor has buffered enough LDBuffers so that one or more of its logical disks is above the minimum write threshold (5MB), the LogicalDiskDistributor will query the write token pool, passing it a set of Writers for which it has enough data. If a write token is available for one of the specified Writers in the set, the pool will return that token, otherwise it will signal that no tokens are available. The LogicalDiskDistributor is required to pass a token for the target Writer along with its LDBuffer list to the next stage, This simple mechanism prevents any Writer's work queue from growing longer than its "fair share" of the available WriterBuffers and provides reverse feedback in the pipeline without adding any new architectural features.

4.6 System Call Behavior

In the construction of any large system, there are always idiosyncrasies in performance that must be identified and corrected. For example, we noticed that the sizes of arguments to Linux `write()` system calls had a dramatic impact on their latency; issuing many small writes per buffer often yielded more performance than issuing a single large write. One would imagine that providing more information about the application's intended behavior to the operating system would result in better management of underlying resources and latency but in this case, the opposite seems to be true. While we are still unsure of the cause of this behavior, it illustrates that the performance characteristics of operating system services can be unpredictable and counter-intuitive.

5 Evaluation

We now evaluate TritonSort's performance and scalability under various hardware configurations.

5.1 Evaluation Environment

We evaluated TritonSort on a 52 node cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24 GB of memory, and 16 500GB 7,200 RPM 2.5" SATA drives. Each hard drive is configured with a single XFS partition. Each XFS partition is configured with a single allocation group to prevent file fragmentation across allocation groups, and is mounted with the `noatime`, `attr2`, `nobarrier`, and `noquota` flags set. Each server has two HP P410 drive controllers with 512MB on-board cache, as well as a Myricom 10 Gbps network interface. The network interconnect we use is a 52-port Cisco Nexus 5020 datacenter switch. The servers run Linux 2.6.35.1, and our implementation of TritonSort is written in C++.

5.2 Comparison to Alternatives

The 100TB Indy GraySort benchmark was introduced in 2009, and hence there are few systems against which we can compare TritonSort's performance. The most recent holder of the Indy GraySort benchmark, DEMSort [18], sorted slightly over 100TB of data on 195 nodes at a rate of 564 GB per minute. TritonSort currently sorts 100TB of data on 52 nodes at a rate of 916 GB per minute, a factor of six improvement in per-node efficiency.

Intermediate Disk Speed (RPM)	Logical Disks Per Physical Disk	Phase 1 Throughput (MBps)	Phase 1 Bottleneck Stage	Average Write Size (MB)
7200	315	69.81	Writer	12.6
7200	158	77.89	Writer	14.0
15000	158	79.73	LogicalDiskDistributor	5.02

Table 2: Effect of increasing speed of intermediate disks on a two node, 500GB sort

5.3 Examining Changes in Balance

We next examine the effect of changing the cluster's configuration to support more memory or faster disks. Due to budgetary constraints, we could not evaluate these hardware configurations at scale. Evaluating the performance benefits of SSDs is the subject of future work.

In the first experiment, we replaced the 500GB, 7200RPM disks that are used as the intermediate disks in phase one and the input disks in phase two with 146GB, 15000RPM disks. The reduced capacity of the drives necessitated running an experiment with a smaller input data set. To allow space for the logical disks to be pre-allocated on the intermediate disks without overrunning the disks' capacity, we decreased the number of logical disks per physical disk by a factor of two. This doubles the amount of data in each logical disk, but the experiment's input data set is small enough that the amount of data per logical disk does not overflow the logical disk's maximum size.

Phase one throughput in these experiments is slightly lower than in subsequent experiments because the 30-35 seconds it takes to write the last few bytes of each logical disk at the end of the phase is roughly 10% of the total runtime due to the relatively small dataset size.

The results of this experiment are shown in Table 2. We first examine the effect of decreasing the number of logical disks without increasing disk speed. Decreasing the number of logical disks increases the average length of LDBuffer chains formed by the LogicalDiskDistributor; note that most of the time, full WriterBuffers (14MB) are written to the disks. In addition, halving the number of logical disks decreases the number of external cylinders that the logical disks occupy, decreasing maximal seek latency. These two factors combine together to net a significant (11%) increase in phase one throughput.

The performance gained by writing to 15000 RPM disks in phase one is much less pronounced. The main reason for this is that the increase in write speed causes the Writers to become fast enough that the LogicalDiskDistributor exposes itself as the bottleneck stage. One side-effect of this is that the LogicalDiskDistributor cannot populate WriterBuffers as fast as they become available, so it reverts to a pathological case in which it always is able to successfully retrieve a write token and hence continuously writes minimally-filled (5MB) buffers. Creating a LogicalDiskDistributor stage that dynamically adjusts its write size based on write token retrieval success rate is the subject of future work.

RAM Per Node (GB)	Phase 1 Throughput (MBps)	Average Write Size (MB)
24	73.53	12.43
48	76.45	19.21

Table 3: Effect of increasing the amount of memory per node on a two node, 2TB sort

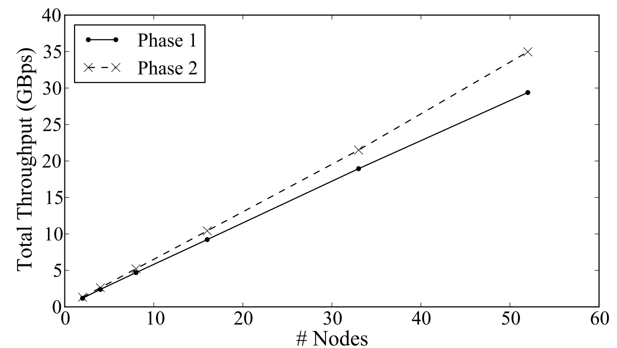


Figure 11: Throughput when sorting 1 TB per node as the number of nodes increases

In the next experiment, we doubled the RAM in two of the machines in our cluster and adjusted TritonSort's memory allocation by doubling the size of each WriterBuffer (from 14MB to 28MB) and using the remaining memory (22GB) to create additional LDBuffers. As shown in Table 3, increasing the amount of memory allows for the creation of longer chains of LDBuffers in the LogicalDiskDistributor, which in turn causes write sizes to increase. The increase in write size is not linear in the amount of RAM added; this is likely because we are approaching the point past which larger writes will not dramatically improve write throughput.

5.4 TritonSort Scalability

Figure 11 shows TritonSort's total throughput when sorting 1 TB per node as the number of nodes increases from 2 to 48. Phase two exhibits practically linear scaling, which is expected since each node performs phase two in isolation. Phase one's scalability is also nearly linear; the slight degradation in its performance at large scales

is likely due to network variance that becomes more pronounced as the number of nodes increases.

6 Discussion and Future Work

In this section, we discuss our system and present directions for future work.

6.1 Supporting More General Sorting

Two assumptions that we make in our design are that tuples are uniform in size, and that they are uniformly and identically distributed across the input files. TritonSort can be extended to support non-uniform tuple sizes by extending the tuple data structure to keep key and value lengths. The most major modification that this will necessitate will be supporting the in-memory sort of keys in phase two, which will require modifications to the phase two Sorter stage. To support the non-uniform distribution of keys across input files, we plan to implement a new phase that will operate before TritonSort begins in which a random small subset of the input data is scanned, determining a histogram of the key distribution. Using this empirical distribution, we will determine a hash function that spreads tuples across nodes as uniformly as possible.

6.2 Automated Performance Tuning

In the current TritonSort prototype, the sizes of individual buffers, the number of buffers of each type, and the number of workers implementing each stage are determined manually. Key to supporting more general hardware configurations and more general DISC applications is the ability to determine these quantities automatically and dynamically. This automatic selection will need to be performed both statically at design time, and dynamically during runtime based on observed conditions. A stage's performance on synthetic data in isolation provides a good upper-bound on its real performance and makes choosing between different implementations easier, but any such synthetic analysis does not take runtime conditions such as CPU scheduling and cache contention into account. Therefore, some manner of online learning algorithm will likely be necessary for the system to determine a good configuration at scale.

6.3 Incorporating SSDs into TritonSort

To achieve nearly sequential-speed throughput to the disks, writes must be large. However, limited per-node memory capacity and high memory cost makes it hard to allocate more than 25MB of memory to each WriterBuffer. Here, we discuss a possible use of SSDs to provide high write speeds with much smaller buffers.

If we were to add three 80GB SSDs to each machine, we could setup a pipeline in which these SSDs are divided between the eight Writers, so that each Writer has 30 GB of SSD space. The LogicalDiskDistributor passes data for each logical disk to the Writer stage in small chunks, where Writers write them to the SSDs. Assuming 315 logical disks per Writer, this gives each logical disk 95 MB of space on the SSD. Because the SSD can handle such a large number of IOPS, there is no penalty for small writes as there is with standard hard drives. Once 80 MB of data is written to a single logical disk on the SSDs, the Writer initiates a *sendfilev()* system call that causes a sequential DMA transfer of that data from the SSD to the appropriate intermediate disk. This should lower our memory requirements to 24 GB, while permitting extremely large writes. This approach relies on two features: significant PCI bandwidth to support parallel transfers to the SSDs, and an SSD array present in the node able to provide high streaming bandwidth to the SSDs; we will need such an array to simultaneously support over 640 MBps of parallel writes and 640 MBps of parallel reads to fully utilize the disks.

7 Related Work

The Datamation sorting benchmark[5] initially measured the elapsed time to sort one million records from disk to disk. As hardware has improved, the number of records has grown to its current level of 100TB. Over the years, numerous authors have reported the performance of their sorting systems, and we benefit from their insights[18, 15, 21, 6, 17, 16]. We differ from previous sort benchmark holders in that we focus on maximizing both aggregate throughput and per-node efficiency.

Achieving per-resource balance in a large-scale data processing system is the subject of a large volume of previous research dating back at least as far as 1970. Among the more well-known guidelines for building such systems are the Amdahl/Case rules of thumb for building balanced systems [3] and Gray and Putzolu's "five-minute rule" [13] for trading off memory and I/O capacity. These guidelines have been re-evaluated and refreshed as hardware capabilities have increased.

NOWSort[6] was the first of the aforementioned sorting systems to run on a shared-nothing cluster. NOWSort employs a two-phase pipeline that generates multiple sorted runs in the first phase and merges them together in the second phase, a technique shared by DEMSort[18]. An evaluation of NOWSort done in 1998[7] found that its performance was limited by I/O bus bandwidth and poor instruction locality. Modern PCI buses and multicore processors have largely eliminated these concerns; in practice, TritonSort is bottlenecked by disk bandwidth.

TritonSort's staged, pipelined dataflow architecture is inspired in part by SEDA[20], a staged, event-driven software architecture that decouples worker stages by interposing queues between them. Other DISC systems such as Dryad [14] export a similar model, although

Dryad has fault-tolerance and data redundancy capabilities that TritonSort does not currently implement.

We are further informed by lessons learned from parallel database systems. Gamma[10] was one of the first parallel database systems to be deployed on a shared-nothing cluster. To maximize throughput, Gamma employs horizontal partitioning to allow separable queries to be performed across many nodes in parallel, an approach that is similar in many respects to our use of logical disks. TritonSort's Sender-Receiver pair is similar to the exchange operator first introduced by Volcano[12] in that it abstracts data partitioning, flow control, parallelism and data distribution from the rest of the system.

8 Conclusions

In this work, we describe the hardware and software architecture necessary to build TritonSort, a highly efficient, pipelined, stage-driven sorting system designed to sort tens to hundreds of TB of data. Through careful management of system resources to ensure cross-resource balance, we are able to sort tens of GB of data per node per minute, resulting in 916 GB/min across only 52 nodes. We believe the work holds a number of lessons for balanced system design and for scale-out architectures in general and will help inform the construction of more balanced data processing systems that will bridge the gap between scalability and per-node efficiency.

9 Acknowledgments

This project was supported by NSF's Center for Integrated Access Networks and NSF MRI #CNS-0923523. We'd like to thank Cisco Systems for their support of this work. We'd like to acknowledge Stefan Savage for providing valuable feedback concerning network optimizations, and thank our shepherd Andrew Birrell and the anonymous reviewers for their feedback and suggestions.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *CACM*, 1988.

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[3] G. Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, 1970.

[4] E. Anderson and J. Tucek. Efficiency matters! In *HotStorage*, 2009.

[5] Anon et al. A Measure of Transaction Processing Power. *Datamation*, 1985.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD*, 1997.

[7] R. Arpaci-Dusseau, A. Arpaci-Dusseau, D. Culler, J. Hellerstein, and D. Patterson. The architectural costs of streaming I/O: A comparison of workstations, clusters, and SMPs. In *HPCA*, pages 90–101, 1998.

[8] R. E. Bryant. Data-Intensive Supercomputing: The Case for DISC. Technical Report CMU-CS-07-128, CMU, 2007.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[10] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *TKDE*, 1990.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, 2003.

[12] G. Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 1994.

[13] J. Gray and G. R. Putzolu. The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. In *SIGMOD*, 1987.

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[15] B. C. Kuszmaul. TeraByte TokuSampleSort, 2007. http://sortbenchmark.org/tokutera.pdf.

[16] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. Lomet. Alphasort: A cache-sensitive parallel external sort. In *VLDB*, 1995.

[17] C. Nyberg, C. Koester, and J. Gray. NSort: a Parallel Sorting Program for NUMA and SMP Machines, 1997.

[18] M. Rahn, P. Sanders, J. Singler, and T. Kieritz. DEMSort – Distributed External Memory Sort, 2009. http://sortbenchmark.org/demsort.pdf.

[19] Sort Benchmark Home Page. http://sortbenchmark.org/.

[20] M. Welsh, D. Culler, and E. Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *SOSP*, 2001.

[21] J. Wyllie. Sorting on a Cluster Attached to a Storage-Area Network, 2005. http://sortbenchmark.org/2005_SCS_Wyllie.pdf.

[22] Apache hadoop. http://hadoop.apache.org/.

[23] Scaling Hadoop to 4000 nodes at Yahoo! http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

Diagnosing performance changes by comparing request flows

Raja R. Sambasivan⋆, Alice X. Zheng†, Michael De Rosa‡, Elie Krevat⋆,
Spencer Whitman⋆, Michael Stroucken⋆, William Wang⋆, Lianghong Xu⋆, Gregory R. Ganger⋆
⋆Carnegie Mellon University, †Microsoft Research, ‡Google

Abstract

The causes of performance changes in a distributed system often elude even its developers. This paper develops a new technique for gaining insight into such changes: comparing request flows from two executions (e.g., of two system versions or time periods). Building on end-to-end request-flow tracing within and across components, algorithms are described for identifying and ranking changes in the flow and/or timing of request processing. The implementation of these algorithms in a tool called Spectroscope is evaluated. Six case studies are presented of using Spectroscope to diagnose performance changes in a distributed storage service caused by code changes, configuration modifications, and component degradations, demonstrating the value and efficacy of comparing request flows. Preliminary experiences of using Spectroscope to diagnose performance changes within select Google services are also presented.

1 Introduction

Diagnosing performance problems in distributed systems is difficult. Such problems may have many sources and may be contained in any one or more of the component processes or, more insidiously, may emerge from the interactions among them [21]. A suite of debugging tools is needed to help in identifying and understanding the root causes of the diverse types of performance problems that can arise. In contrast to single-process applications, for which diverse performance debugging tools exist (e.g., DTrace [6], gprof [14], and GDB [12]), too few techniques have been developed for guiding diagnosis of distributed system performance.

Recent research has developed promising new techniques that can help populate the suite. Many build on low-overhead end-to-end tracing (e.g., [4, 7, 9, 11, 31, 34]), which captures the *flow* (i.e., path and timing) of individual requests within and across the components of a distributed system. For example, with such rich information about a system's operation, researchers have developed new techniques for detecting anomalous request flows [4], spotting large-scale departures from performance models [33], and comparing observed behaviour to manually-constructed expectations [26].

This paper develops a new technique for the suite: comparing request flows between two executions to identify why performance has changed between them. Such comparison allows one execution to serve as a model of acceptable performance; highlighting key differences from this model and understanding their performance costs allows for easier diagnosis than when only a single execution is used. Though obtaining an execution of acceptable performance may not be possible in all cases—e.g., when a developer wants to understand why performance has always been poor—there are many cases for which request-flow comparison is useful. For example, it can help diagnose performance changes resulting from modifications made during software development (e.g., during regular regression testing) or from upgrades to components of a deployed system. Also, it can help when diagnosing changes over time in a deployed system, which may result from component degradations, resource leakage, or workload changes.

Our analysis of bug tracking data for a distributed storage service indicates that more than half of the reported performance problems would benefit from guidance provided by comparing request flows. Talks with Google engineers [3] and experiences using request-flow comparison to diagnose Google services affirm its utility.

The utility of comparing request flows relies on the observation that performance changes often manifest as changes in how requests are serviced. When comparing two executions, which we refer to as the *non-problem period* (before the change) and the *problem period* (after the change), there will usually be some changes in the observed request flows. We refer to new request flows in the problem period as *mutations* and to the request flows corresponding to how they were serviced in the non-problem period as *precursors*. Identifying mutations and comparing them to their precursors helps localize sources of change and gives insight into their effects.

This paper describes algorithms for effectively comparing request flows across periods, including for identifying mutations, ranking them based on their contribution to the overall performance change, identifying their most likely precursors, highlighting the most prominent divergences, and identifying low-level parameter differences that most strongly correlate to each.

We categorize mutations into two types: *Response-time mutations* correspond to requests that have increased only in cost between the periods; their precursors are requests that exhibit the same structure, but whose response time is different. *Structural mutations* correspond to requests that take different paths through the system in the problem period. Identifying their precursors requires analysis of all request flows with differing frequencies in the two periods.

Figure 1 illustrates a (mocked up) example of two mutations and their precursors. Ranking and highlighting divergences involves using statistical tests and comparison of mutations and associated precursors.

We have implemented request-flow comparison in a toolset called Spectroscope and used it to diagnose performance problems observed in Ursa Minor [1], a distributed storage service. By describing five real problems and one synthetic one, we illustrate the utility of comparing request flows and show that our algorithms enable effective use of this technique. To understand challenges associated with scaling request-flow comparison to very large distributed systems, this paper also describes preliminary experiences using it to diagnose performance changes within distributed services at Google.

2 End-to-end request-flow tracing

Request-flow comparison builds on end-to-end tracing, an invaluable information source that captures a distributed system's performance and control flow in detail. Such tracing works by capturing *activity records* at each of various trace points within the distributed system's software, with each record identifying the specific tracepoint name, the current time, and other contextual information. Most implementations associate activity records with individual requests by propagating a per-request identifier, which is stored within the record. Activity records can be stitched together, either offline or online, to yield request-flow graphs, which show the control flow of individual requests. Several efforts, including Magpie [4], Whodunit [7], Pinpoint [9], X-Trace [10, 11], Google's Dapper [31], and Stardust [34] have independently implemented such tracing and shown that it can be used continuously with low overhead, especially when request sampling is supported [10, 28, 31]. For example, Stardust [34], Ursa Minor's end-to-end tracing mechanism, adds 1% or less overhead when used with key benchmarks, such as SpecSFS [30].

End-to-end tracing implementations differ in two key respects: whether instrumentation is added automatically or manually and whether the request flows can disambiguate sequential and parallel activity. With regard to the latter, Magpie [4] and recent versions of both Stardust [34] and X-Trace [10] explicitly account for concurrency by embedding information about thread synchronization in their traces (see Figure 2). These implementations are a natural fit for request-flow comparison, as they can disambiguate true structural differences from false ones caused by alternate interleavings of concurrent activity. Whodunit [7], Pinpoint [9], and Dapper [31] do not account for parallelism.

End-to-end tracing in distributed systems is past the research stage. For example, it is used in production Google datacenters [31] and in some production three-tier systems [4]. Research continues, however, on how to best exploit the information provided by such tracing.

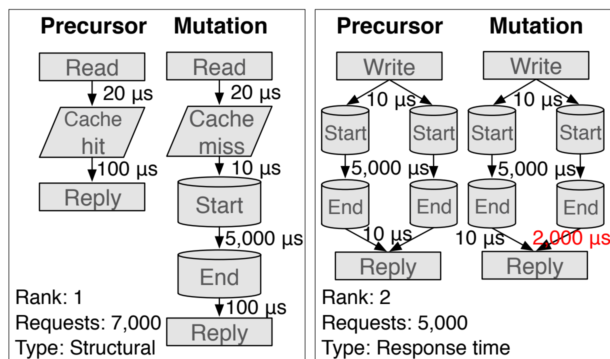


Figure 1: Example output from comparing request flows. The two mutations shown are ranked by their effect on the change in performance. The item ranked first is a structural mutation and the item ranked second is a response-time mutation. Due to space constraints, mocked-up graphs are shown in which nodes represent the type of component accessed.

3 Behavioural changes vs. anomalies

Our technique of comparing request flows between two periods identifies distribution changes in request-flow behaviour and ranks them according to their contribution to the observed performance difference. Conversely, anomaly detection techniques, as implemented by Magpie [4] and Pinpoint [9], mine a single period's request flows to identify rare ones that differ greatly from others. In contrast to request-flow comparison, which attempts to identify the most important differences between two sets, anomaly detection attempts to identify rare elements within a single set.

Request-flow comparison and anomaly detection serve distinct purposes, yet both are useful. For example, performance problems caused by changes in the components used (e.g., see Section 8.2), or by common requests whose response times have increased slightly, can be easily diagnosed by comparing request flows, whereas many anomaly detection techniques will be unable to provide guidance. In the former case, guidance will be difficult because the changed behaviour is common during the problem period; in the latter, because the per-request change is not extreme enough.

4 Spectroscope

To illustrate the utility of comparing request flows, this technique was implemented in a tool called Spectroscope and used to diagnose performance problems seen in Ursa Minor [1] and in certain Google services. This section provides an overview of Spectroscope, and the next describes its algorithms. Section 4.1 describes how *cate-*

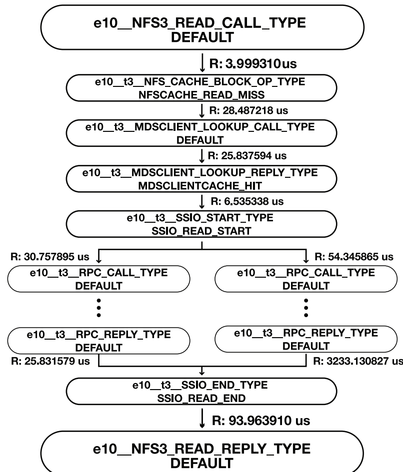


Figure 2: Example request-flow graph. The graph shows a striped READ in the Ursa Minor distributed storage system. Nodes represent trace points and edges are labeled with the time between successive events. Parallel substructures show concurrent threads of activity. Node labels are constructed by concatenating the machine name (e.g., e10), component name (e.g., NFS3), trace-point name (e.g., READ_CALL_TYPE), and an optional semantic label (e.g., NFSCACHE_READ_MISS). Due to space constraints, trace points executed on other components as a result of the NFS server's RPC calls are not shown.

gories, the basic building block on which Spectroscope operates, are constructed. Section 4.2 describes Spectroscope's support for comparing request flows.

4.1 Categorizing request flows

Even small distributed systems can service hundreds to thousands of requests per second, so comparing all of them individually is not feasible. Instead, exploiting a general expectation that requests that take the same path should incur similar costs, Spectroscope groups identically-structured requests into unique *categories* and uses them as the basic unit for comparing request flows. For example, requests whose structures are identical because they hit in a NFS server's data and metadata cache will be grouped into the same category, whereas requests that miss in both will be grouped in a different one. Two requests are deemed structurally identical if their string representations, as determined by a depth-first traversal, are identical. For requests with parallel substructures, Spectroscope computes all possible string representations when determining the category in which to bin them. The exponential cost is mitigated by imposing an order on parallel substructures (i.e., by always traversing them in alphabetical order, as determined by their root node names) and by the fact that parallelism is limited in most request flows we have observed.

For each category, Spectroscope identifies aggregate statistics, including request count, average response time, and variance. To identify where time is spent, it also computes average edge latencies and corresponding variances. Spectroscope displays categories in either a graph view, with statistical information overlaid, or within train-schedule visualizations [37] (also known as swim lanes), which more directly show the constituent requests' pattern of activity.

Spectroscope uses selection criteria to limit the number of categories developers must examine. For example, when comparing request flows, statistical tests and a ranking scheme are used. The number of categories could be further reduced by using unsupervised clustering algorithms, such as those used in Magpie [4], to bin similar but not necessarily identical requests into the same category. Initial versions of Spectroscope used off-the-shelf clustering algorithms [29], but we found the groups they created too coarse-grained and unpredictable. Often, they would group mutations and precursors within the same category, masking their existence. For clustering algorithms to be useful, improvements such as distance metrics that better align with developers' notions of request similarity are needed. Without them, use of clustering algorithms will result in categories composed of seemingly dissimilar requests.

4.2 Comparing request flows

Performance changes can result from a variety of factors, such as internal changes to the system that result in performance regressions, unintended side effects of changes to configuration files, or environmental issues. Spectroscope helps diagnose these problems by comparing request flows and identifying the key resulting mutations. Figure 3 shows Spectroscope's workflow.

When comparing request flows, Spectroscope takes as input request-flow graphs from two periods of activity, which we refer to as a *non-problem period* and a *problem period*. It creates categories composed of requests from both periods and uses statistical tests and heuristics to identify which contain structural mutations, response-time mutations, or precursors. Categories containing mutations are presented to the developer in a list ranked by expected contribution to the performance change. Note that the periods do not need to be aligned exactly with the performance change (e.g., at Google we often chose day-long periods based on historic average latencies).

Visualizations of categories that contain mutations are similar to those described previously, except per-period statistical information is shown. The root cause of response-time mutations is localized by showing the edges responsible for the mutation in red. The root cause of structural mutations is localized by providing a ranked list of the candidate precursors, so that the developer can determine how they differ. Figure 1 shows an example.

Spectroscope provides further insight into perfor-

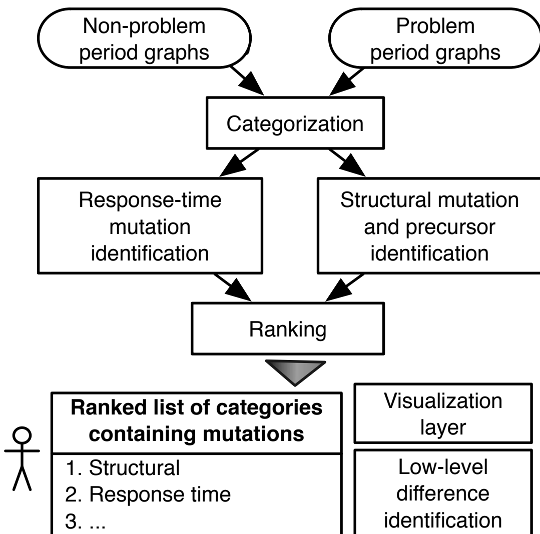


Figure 3: Spectroscope's workflow for comparing request flows. First, Spectroscope groups requests from both periods into categories. Second, it identifies which categories contain mutations or precursors. Third, it ranks mutation categories according to their expected contribution to the performance change. Developers are presented this ranked list. Visualizations of mutations and their precursors can be shown. Also, low-level differences can be identified for them.

mance changes by identifying the low-level parameters (e.g., client parameters or function call parameters) that best differentiate a chosen mutation and its precursors. For example, in Ursa Minor, one performance slowdown, which manifested as many structural mutations, was caused by a change in a parameter sent by the client. For problems like this, highlighting the specific low-level differences can immediately identify the root cause.

Section 5 describes Spectroscope's algorithms and heuristics for identifying mutations, their corresponding precursors, their rank based on their relative influence on the overall performance change, and their most relevant low-level parameter differences. It also describes how these methods overcome key challenges—for example, differentiating true mutations from natural variance in request structure and timings. Identification of response-time mutations and ranking rely on the expectation (reasonable for many distributed systems, including distributed storage) that requests that take the same path through a distributed system will exhibit similar response times and edge latencies. Section 7 describes how high variance in this axis affects Spectroscope's results.

5 Algorithms for comparing request flows

This section describes the key heuristics and algorithms used when comparing request flows. In creating them, we favoured simplicity and those that regulate false positives—perhaps the worst failure mode due to developer effort wasted—whenever possible.

5.1 Identifying response-time mutations

When comparing two periods, there will always be some natural differences in timings. Spectroscope uses the Kolmogorov-Smirnov two-sample, non-parametric hypothesis test [20] to differentiate natural variance from true changes in distribution or behaviour. Statistical hypothesis tests take as input two distributions and output a p-value, which represents uncertainty in the claim that the null hypothesis, that both distributions are the same, is false. Expensive false positives are limited to a preset rate (almost always 5%) by rejecting the null hypothesis only when the p-value is lower than this value. The p-value increases with variance and decreases with the number of samples. A non-parametric test, which does not require knowledge of the underlying distribution, is used because we have observed that response times are not governed by well-known distributions.

The Kolmogorov-Smirnov test is used as follows. For each category, the distributions of response times for the non-problem period and the problem period are extracted and input into the hypothesis test. The category is marked as containing response-time mutations if the test rejects the null hypothesis. By default, categories that contain too few requests to run the test accurately are not marked as containing mutations. To identify the components or interactions responsible for the mutation, Spectroscope extracts the critical path—i.e., the path of the request on which response time depends—and runs the same hypothesis test on the edge latency distributions. Edges for which the null hypothesis is rejected are marked in red in the final output visualization.

5.2 Identifying structural mutations

To identify structural mutations, Spectroscope assumes a similar workload was run in both the non-problem period and the problem period. As such, it is reasonable to expect that an increase in the number of requests that take one path through the distributed system in the problem period should correspond to a decrease in the number of requests that take other paths. Since non-determinism in service order dictates that per-category counts will always vary slightly between periods, a threshold is used to identify categories that contain structural mutations and precursors. Categories that contain `SM_THRESHOLD` more requests from the problem period than from the non-problem period are labeled as containing mutations and those that contain `SM_THRESHOLD` fewer are labeled as containing precursors.

Choosing a good threshold for a workload may require some experimentation, as it is sensitive to both the number of requests issued and the sampling rate. Fortunately,

it is easy to run Spectroscope multiple times, and it is not necessary to get the threshold exactly right—choosing a value that is too small will result in more false positives, but they will be given a low rank and so will not mislead the developer in his diagnosis efforts.

If per-category distributions of request counts are available, a statistical test, instead of a threshold, could be used to determine those categories that contain mutations or precursors. This statistical approach would be superior to a threshold-based approach, as it guarantees a set false-positive rate. However, building the distributions necessary would require obtaining many non-problem and problem-period datasets, so we opted for the simpler threshold-based approach instead. Also, our experiences at Google indicate that request structure within large datacenters may change too quickly for such expensive-to-build models to be useful.

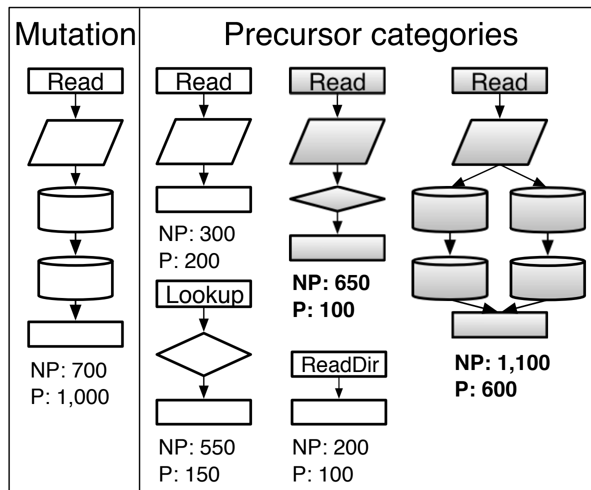


Figure 4: How the precursor categories of a structural-mutation category are identified. One structural-mutation category and five precursor categories are shown, each with their corresponding request counts from the non-problem (NP) and problem (P) periods. For this case, the shaded precursor categories will be identified as those that could have donated requests to the structural-mutation category. The precursor categories that contain LOOKUP and READDIR requests cannot have donated requests because their constituent requests are not READS. The top left-most precursor category contains READS, but the 1:N heuristic eliminates it.

5.3 Mapping mutations to precursors

Once the total set of categories that contain structural mutations and precursors has been identified, Spectroscope must iterate through each structural-mutation category to determine the precursor categories that are likely to have donated requests to it. This is accomplished via three heuristics, described below. Figure 4 shows how they are applied.

First, the total list of precursor categories is pruned to eliminate categories with a different root node than those in the structural-mutation category. The root node describes the overall type of a request, for example READ, WRITE, or READDIR, and requests of different high-level types should not be precursor/mutation pairs.

Second, remaining precursor categories that have decreased in request count less than the increase in request count of the structural-mutation category are also removed from consideration. This 1:N heuristic reflects the common case that one precursor category is likely to donate requests to N structural-mutation categories. For example, a cache-related problem may result in a portion of requests that used to hit in that cache to miss and hit in the next-level cache. Extra cache pressure at this next-level cache may result in the rest missing in both caches. This heuristic can be optionally disabled.

Third, the remaining precursor categories are ranked according to their likelihood of having donated requests, as determined by the string-edit distance between them and the structural-mutation category. This heuristic reflects the intuition that precursors and structural mutations are likely to resemble each other in structure. The cost of computing the edit distance is $O(NM)$, where N and M are the lengths of the string representations of the categories being compared.

5.4 Ranking

Ranking of mutations is necessary for two reasons. First, the performance problem might have multiple root causes, each of which causes its own set of mutations. Second, even if there is only one root cause to the problem (e.g., a misconfiguration), many mutations will often still be observed. For both cases, it is useful to identify the mutations that most affect performance in order to focus diagnosis effort where it will yield the most benefit.

Spectroscope ranks categories that contain mutations in descending order by their expected contribution to the performance change. The contribution for a structural-mutation category is calculated as the number of mutations it contains, which is the difference between its problem and non-problem period counts, multiplied by the difference in problem period average response time between it and its precursor categories. If more than one candidate precursor category has been identified, a weighted average of their average response times is used; weights are based on structural similarity to the mutation. The contribution for a response-time-mutation category is calculated as the number of mutations it contains, which is just the non-problem period count, times the change in average response time of that category be-

tween the periods. If a category contains both response-time mutations and structural mutations, it is split into two virtual categories and each is ranked separately.

5.5 Identifying low-level differences

Identifying the differences in low-level parameters between a mutation and precursor can often help developers further localize the source of the problem. For example, the root cause of a response-time mutation might be further localized by identifying that it is caused by a component that is sending more data in its RPCs than during the non-problem period.

Spectroscope allows developers to pick a mutation category and candidate precursor category for which to identify low-level differences. Given these categories, Spectroscope induces a regression tree [5] showing the low-level parameters that best separate requests in these categories. Each path from root to leaf represents an independent explanation of why the mutation occurred. Since developers may already possess some intuition about what differences are important, the process is meant to be interactive. If the developer does not like the explanations, he can select a new set by removing the root parameter from consideration and re-running the algorithm.

The regression tree is induced as follows. First, a depth-first traversal is used to extract a template describing the parts of request structures that are common between both categories, up until the first observed difference. Portions that are not common are excluded, since low-level parameters cannot be compared for them.

Second, a table in which rows represent requests and columns represent parameters is created by iterating through each of the categories' requests and extracting parameters from the parts that fall within the template. Each row is labeled as belonging to the problem or non-problem period. Certain parameter values, such as the `thread ID` and `timestamp`, must always be ignored, as they are not expected to be similar across requests. Finally, the table is fed as input to the C4.5 algorithm [25], which creates the regression tree. To reduce the runtime, only parameters from a randomly sampled subset of requests are extracted from the database, currently a minimum of 100 and a maximum of 10%. Parameters only need to be extracted the first time the algorithm is run; subsequent iterations can modify the table directly.

5.6 Current limitations

This section describes current limitations with our techniques for comparing request flows.

Diagnosing problems caused by contention: Our techniques assume that performance changes are caused by changes to the system (code changes, configuration changes, etc). Though they will identify mutations caused by contention, they cannot directly attribute them to the responsible process. In some cases our techniques can indirectly help—for example, by showing that many edges within a component are responsible for a response-time mutation, they can help the developer intuit that the problem is due to contention with an external process.

Diagnosing problems when the load differs significantly between periods: In such cases, the load change itself may be the root cause. Though our techniques will identify response-time and structural changes when the load during the problem period is much greater than the non-problem period, the developer must determine whether they are reasonable degradations.

6 Experimental apparatus

Most of the experiments and case studies reported in this paper come from using Spectroscope with a distributed storage service called Ursa Minor. Section 6.1 describes this system. Section 6.2 describes the benchmarks used for Ursa Minor's nightly regression tests, the setting in which many of the case studies were observed.

To understand issues in scaling request-flow comparison to larger systems, we also used Spectroscope to diagnose services within Google. Section 6.3 provides more details. The implementation of Spectroscope for Ursa Minor was written in Perl and MATLAB. It includes a visualization layer built upon Prefuse [16]. The cost of calculating edit distances dominates its runtime, so it is sensitive to the value of `SM_THRESHOLD` used. The implementation for Google was written in C++; its runtime is much lower (on the order of seconds) and its visualization layer uses DOT [15].

6.1 Ursa Minor

Figure 5 illustrates Ursa Minor's architecture. Like most modern scalable distributed storage, Ursa Minor separates metadata services from data services, such that clients can access data on storage nodes without moving it all through metadata servers. An Ursa Minor instance (called a "constellation") consists of potentially many NFS servers (for unmodified clients), storage nodes (SNs), metadata servers (MDSs), and end-to-end-trace servers. To access data, clients must first send a request to a metadata server asking for the appropriate permissions and locations of the data on the storage nodes. Clients can then access the storage nodes directly.

Ursa Minor has been in active development since 2004 and comprises about 230,000 lines of code. More than 20 graduate students and staff have contributed to it over its lifetime. More details about its implementation can be found in Abd-El-Malek et al. [1].

The components of Ursa Minor are usually run on separate machines within a datacenter. Though Ursa Minor supports an arbitrary number of components, the experi-

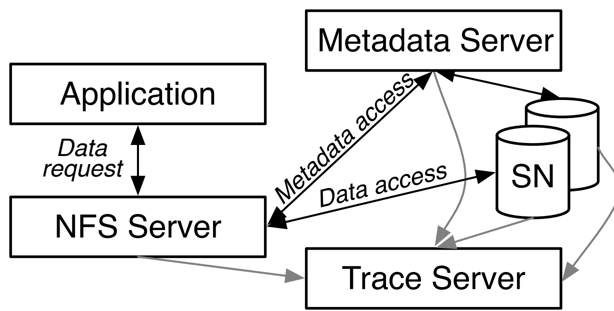


Figure 5: Ursa Minor Architecture. Ursa Minor can be deployed in many configurations, with an arbitrary number of NFS servers, metadata servers, storage nodes (SNs), and trace servers. Here, a simple five-component configuration is shown.

ments and case studies detailed in this paper use a simple five-machine configuration: one NFS server, one metadata server, one trace server, and two storage nodes. One storage node stores data, while the other stores metadata. Not coincidentally, this is the configuration used in the nightly regression tests that uncovered many of the problems described in the case studies.

End-to-end tracing infrastructure via Stardust: Ursa Minor's Stardust tracing infrastructure is much like its peer group, discussed in Section 2. Request sampling is used to capture trace data for a subset of entire requests (10% by default), with a per-request decision made randomly when the request enters the system. Ursa Minor contains approximately 200 trace points, 124 manually inserted as well as automatically generated ones for each RPC send and receive function. In addition to simple trace points, which indicate points reached in the code, explicit split and join trace points are used to identify the start and end of concurrent threads of activity. Low-level parameters are also collected at trace points.

6.2 Benchmarks used with Ursa Minor

Experiments run on Ursa Minor use these benchmarks.

`Linux-build` and `ursa minor-build`: These benchmarks consist of two phases: a copy phase, in which the source tree is tarred and copied to Ursa Minor and then untarred, and a build phase, in which the source files are compiled. `Linux-build` (of 2.6.32 kernel) runs for 26 minutes. About 145,000 requests are sampled. The average graph size and standard deviation is 12 and 40 nodes. Most graphs are small, but some are very big, so the per-category equivalents are larger: 160 and 500 nodes. `Ursa minor-build` runs for 10 minutes. About 16,000 requests are sampled and the average graph size and standard deviation is 9 and 28 nodes. The per-category equivalents are 96 and 100 nodes.

`Postmark-large`: This synthetic benchmark evaluates the small file performance of storage systems [19]. It utilizes 448 subdirectories, 50,000 transactions, and 200,000 files and runs for 80 minutes. The average graph size and standard deviation is 66 and 65 nodes. The per-category equivalents are 190 and 81 nodes.

`SPEC SFS 97 V3.0 (SFS97)`: This synthetic benchmark is the industry standard for measuring NFS server scalability and performance [30]. It applies a periodically increasing load of NFS operations to a storage system's NFS server and measures the average response time. It was configured to generate load between 50 and 350 operations/second in increments of 50 ops/second and runs for 90 minutes. The average graph size and standard deviation is 30 and 51 nodes. The per-category equivalents are 206 and 200 nodes.

`IoZone`: This benchmark [23] sequentially writes, rewrites, reads, and re-reads a 5GB file in 20 minutes. The average graph size and standard deviation is 6 nodes. The per-category equivalents are 61 and 82 nodes.

6.3 Dapper & Google services

The Google services for which Spectroscope was applied were instrumented using Dapper, which automatically embeds trace points in Google's RPC framework. Like Stardust, Dapper employs request sampling, but uses a sampling rate of less than 0.1%. Spectroscope was implemented as an extension to Dapper's aggregation pipeline, which groups individual requests into categories and was originally written to support Dapper's pre-existing analysis tools. Categories created by the aggregation pipeline only show compressed call graphs with identical children and siblings merged together.

7 Dealing with high-variance categories

For automated diagnosis tools to be useful, it is important that distributed systems satisfy certain properties about variance. For Spectroscope, categories that exhibit high variance in response times and edge latencies do not satisfy the expectation that "requests that take the same path should incur similar costs" and can affect its ability to identify mutations accurately. Spectroscope's ability to identify response-time mutations is sensitive to variance, whereas for structural mutations only the ranking is affected. Though categories may exhibit high variance intentionally (for example, due to a scheduling algorithm that minimizes mean response time at the expense of variance), many do so unintentionally, as a result of latent performance problems. For example, in early versions of Ursa Minor, several high-variance categories resulted from a poorly written hash table that exhibited slowly increasing lookup times because of a poor hashing scheme.

For response-time mutations, both false negatives and false positives will increase with the number of high-variance categories. False negatives will increase

because high variance will reduce the Kolmogorov-Smirnov test's power to differentiate true behaviour changes from natural variance. False positives, which are much rarer, will increase when it is valid for categories to exhibit similar response times within a single period, but different response times across different ones. The rest of this section concentrates on the false negative case.

To quantify how well categories meet the same path/similar costs expectation within a single period, Figure 6 shows a CDF of the squared coefficient of variation in response time (C^2) for *large categories* induced by `linux-build`, `postmark-large`, and `SFS97` in Ursa Minor. Figure 7 shows the same C^2 CDF for large categories induced by Bigtable [8] running in three Google datacenters over a 1-day period. Each Bigtable instance is shared among the machines in its datacenter and services several workloads. C^2 is a normalized measure of variance and is defined as $(\frac{\sigma}{\mu})^2$. Distributions with C^2 less than one exhibit low variance, whereas those with C^2 greater than one exhibit high variance. *Large categories* contain more than 10 requests; Tables 1 and 2 show that they account for only 15–45% of all categories, but contain more than 98% of all requests. Categories containing fewer requests are not included, since their smaller sample size makes the C^2 statistic unreliable for them.

For the benchmarks run on Ursa Minor, at least 88% of the large categories exhibit low variance. C^2 for all the categories generated by `postmark-large` is small. More than 99% of its categories exhibit low variance and the maximum C^2 value observed is 6.88. The results for `linux-build` and `SFS97` are slightly more heavy-tailed. For `linux-build`, 96% of its categories exhibit low variance, and the maximum C^2 value is 394. For `SFS97`, 88% exhibit C^2 less than 1, and the maximum C^2 value is 50.3. Analysis of categories in the large tail of these benchmarks show that part of the observed variance is a result of contention for locks in the metadata server.

The traces collected for Bigtable by Dapper are relatively sparse—often graphs generated for it are composed of only a few nodes, with one node showing the incoming call type (e.g., READ, MUTATE, etc.) and another showing the call type of the resulting GFS [13] request. As such, many dissimilar paths cannot be disambiguated and have been merged together in the observed categories. Even so, 47–69% of all categories exhibit C^2 less than 1. Additional instrumentation, such as those that show the sizes of Bigtable data requests and work done within GFS, would serve to further disambiguate unique paths and considerably reduce C^2.

8 Ursa Minor case studies

Spectroscope is not designed to replace developers; rather it is intended to serve as an important step in the workflow they use to diagnose problems. Sometimes it can help developers identify the root cause immediately, or at least localize the problem to a specific area of the system. In other cases, it can help eliminate the distributed system as the root cause by verifying that its behaviour has not changed, allowing developers to focus their efforts on external factors.

This section presents diagnoses of six performance problems solved by using Spectroscope to compare request flows and analyzes its effectiveness in identifying the root causes. Most of these problems were previously unsolved and diagnosed by the authors without knowledge of the root cause. One problem was observed before Spectroscope was available, so it was re-injected to show how effectively it could have been diagnosed. By introducing a synthetic spin loop of different delays, we also demonstrate Spectroscope's ability to diagnose changes in response time.

8.1 Methodology

Three complementary metrics are provided for evaluating Spectroscope's output.

The percentage of the 10 highest-ranked categories that are relevant: This metric measures the quality of the rankings of the results. It accounts for the fact that developers will naturally investigate the highest-ranked categories first, so it is important for them to be relevant.

The percentage of false-positive categories: This metric evaluates the quality of the ranked list by identifying the percentage of all results that are *not relevant*.

Request coverage: This metric evaluates quality of the ranked list by identifying the percentage of requests affected by the problem that are identified in it.

Table 3 summarizes Spectroscope's performance using these metrics. Unless otherwise noted, a default value of 50 was used for `SM_THRESHOLD`. We chose this value to yield reasonable runtimes (between 15-30 minutes) when diagnosing problems in larger benchmarks, such as `SFS97` and `postmark-large`. When necessary, it was lowered to further explore the space of possible structural mutations.

8.2 MDS configuration change

After a particular large code check-in, performance of `postmark-large` decayed significantly, from 46tps to 28tps. To diagnose this problem, we used Spectroscope to compare request flows between two runs of `postmark-large`, one from before the check-in and one from after. The results showed many categories that contained structural mutations. Comparing them to their most-likely precursor categories revealed that the storage node utilized by the metadata server had changed. Before the check-in, the metadata server wrote metadata only to its dedicated storage node. After the check-in, it issued most writes to the data storage node instead. We

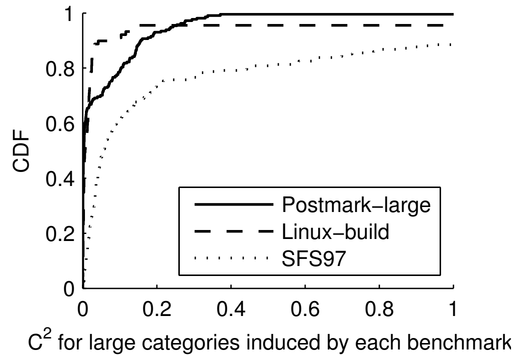


Figure 6: CDF of C^2 for large categories induced by three benchmarks run on Ursa Minor. At least 88% of the categories induced by each benchmark exhibit low variance ($C^2 < 1$). The results for `linux-build` and `SFS` are more heavy-tailed than `postmark-large`, partly due to extra lock contention in the metadata server.

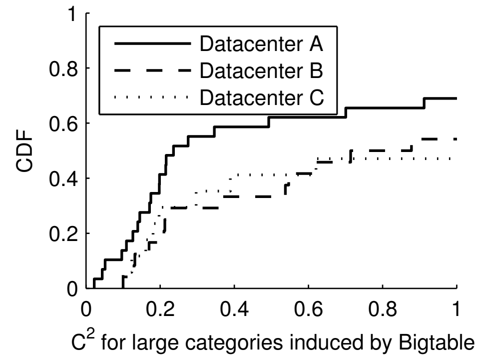


Figure 7: CDF of C^2 for large categories induced by Bigtable instances in three Google datacenters. Dapper's instrumentation of Bigtable is sparse, so many paths cannot be disambiguated and have been merged together in the observed categories, resulting in a higher C^2 than otherwise expected. Even so, 47–69% of categories exhibit low variance.

	Benchmark		
	`Linux-bld`	`Postmark`	`SFS97`
Categories	351	716	1602
Large categories (%)	25.3	29.9	14.7
Requests sampled	145,167	131,113	210,669
In large categories (%)	99.7	99.2	98.9

Table 1: Distribution of requests in the categories induced by three benchmarks run on Ursa Minor. Though many categories are generated, most contain only a small number of requests. *Large categories*, which contain more than 10 requests, account for between 15–29% of all categories generated, but contain over 99% of all requests.

	Google datacenter		
	A	B	C
Categories	29	24	17
Large categories (%)	32.6	45.2	26.9
Requests sampled	7,088	5,556	2,079
In large categories (%)	97.7	98.8	93.1

Table 2: Distribution of requests in the categories induced by three instances of Bigtable over a 1-day period. Fewer categories and requests are observed than for Ursa Minor, because Dapper samples less than 0.1% of all requests. The distribution of requests within categories is similar to Ursa Minor—a small number of categories contain most requests.

also used Spectroscope to identify the low-level parameter differences between a few structural-mutation categories and their corresponding precursor categories. The regression tree found differences in elements of the data distribution scheme (e.g., type of fault tolerance used).

We presented this information to the developer of the metadata server, who told us the root cause was a change in an infrequently-modified configuration file. Along with the check-in, he had mistakenly removed a few lines that pre-allocated the file used to store metadata and specify the data distribution. Without this, Ursa Minor used its default distribution scheme and sent all writes to the data storage node. The developer was surprised to learn that the default distribution scheme differed from the one he had chosen in the configuration file.

Summary: For this real problem, comparing request flows helped developers diagnose a performance change caused by modifications to the system configuration. Many distributed systems contain large configuration files with esoteric parameters (e.g., hadoop-site.xml) that, if modified, can result in perplexing performance changes. Spectroscope can provide guidance in such cases by showing how various configuration options affect system behaviour.

Quantitative analysis: For the evaluation in Table 3, results in the ranked list were deemed relevant if they included metadata accesses to the data storage node with a most-likely precursor category that included metadata accesses to the metadata storage node.

8.3 Read-modify-writes

This problem was observed and diagnosed before development on Spectroscope began; it was re-injected in Ursa Minor to show how Spectroscope could have helped developers easily diagnose it.

A few years ago, performance of `IoZone` declined from 22MB/s to 9MB/s after upgrading the Linux kernel from 2.4.22 to 2.6.16.11. To debug this problem, one of the authors of this paper spent several days manually mining Stardust traces and eventually discovered the

# / Type	Name	Manifestation	Root cause	# of results	Quality of results: Top 10 rel. (%)	FPs (%)	Cov. (%)
8.2 / Real	MDS config.	Structural	Config. change	128	100	2	70
8.3 / Real	RMWs	Structural	Env. change	3	100	0	100
8.4 / Real	MDS prefetch. 50	Structural	Internal change	7	29	71	93
8.4 / Real	MDS prefetch. 10			16	70	56	96
8.5 / Real	Create behaviour	Structural	Design problem	11	40	64	N/A
8.6 / Synthetic	100μs delay	Response time	Internal change	17	0	100	0
8.6 / Synthetic	500μs delay			166	100	6	92
8.6 / Synthetic	1ms delay			178	100	7	93
8.7 / Real	Periodic spikes	No change	Env. change	N/A	N/A	N/A	N/A

Table 3: Overview of the Ursa Minor case studies. This table shows information about each of six problems diagnosed using Spectroscope. For most of the case studies, quantitative metrics that evaluate the quality of Spectroscope's results are included.

root cause: the new kernel's NFS client was no longer honouring the NFS server's preferred READ and WRITE I/O sizes, which were set to 16KB. The smaller I/O sizes used by the new kernel forced the NFS server to perform many *read-modify-write*s (RMWs), which severely affected performance. To remedy this issue, support for smaller I/O sizes was added to the NFS server and counters were added to track the frequency of RMWs.

To show how comparing request flows and identifying low-level parameter differences could have helped developers quickly identify the root cause, Spectroscope was used to compare request flows between a run of `IoZone` in which the Linux client's I/O size was set to 16KB and another during which the Linux client's I/O size was set to 4KB. All of the results in the ranked list were structural-mutation categories that contained RMWs.

We next used Spectroscope to identify the low-level parameter differences between the highest-ranked result and its most-likely precursor category. The output perfectly separated the constituent requests by the `count` parameter, which specifies the amount of data to be read or written by the request. Specifically, requests with `count` parameter values less than or equal to 4KB were classified as belonging to the problem period.

Summary: Diagnosis of this problem demonstrates how comparing request flows can help developers identify performance problems that arise due to a workload change. It also showcases the utility of highlighting relevant low-level parameter differences.

Quantitative analysis: For Table 3, results in the ranked list were deemed relevant if they contained RMWs and their most-likely precursor category did not.

8.4 MDS prefetching

A few years ago, several developers, including one of the authors of this paper, tried to add *server-driven metadata prefetching* to Ursa Minor [17]. This feature was intended to improve performance by prefetching metadata to clients on every mandatory metadata server access, in hopes of minimizing the total number of accesses necessary. However, when implemented, this feature provided no improvement. The developers spent a few weeks (off and on) trying to understand the reason for this unexpected result but eventually moved on to other projects without an answer.

To diagnose this problem, we compared two runs of `linux-build`, one with prefetching disabled and another with it enabled. `linux-build` was chosen because it is more likely to see performance improvements due to prefetching than the other workloads.

When we ran Spectroscope with `SM_THRESHOLD` set to 50, several categories were identified as containing mutations. The two highest-ranked results immediately piqued our interest, as they contained WRITEs that exhibited an abnormally large number of lock acquire/release accesses within the metadata server. All of the remaining results contained response-time mutations from regressions in the metadata prefetching code path, which had not been properly maintained. To further explore the space of structural mutations, we decreased `SM_THRESHOLD` to 10 and re-ran Spectroscope. This time, many more results were identified; most of the highest-ranked ones now exhibited an abnormally high number of lock accesses and differed only in the exact number.

Analysis revealed that the additional lock/unlock calls reflected extra work performed by requests that accessed the metadata server to prefetch metadata to clients. To verify this as the root cause, we added instrumentation around the prefetching function to see whether it caused the database accesses. Altogether, this information provided us with the intuition necessary to determine why server-driven metadata prefetching did not improve performance: the extra time spent in the DB calls by metadata server accesses outweighed the time savings gener-

ated by the increase in client cache hits.

Summary: This problem demonstrates how comparing request flows can help developers account for unexpected performance loss when adding new features. In this case, the problem was due to unanticipated contention several layers of abstraction below the feature addition. Note that diagnosis with Spectroscope is interactive, in this case involving developers iteratively modifying SM_THRESHOLD to gain additional insight.

Quantitative analysis: For Table 3, results in the ranked list were deemed relevant if they contained at least 30 LOCK_ACQUIRE → LOCK_RELEASE edges. Results for the output when SM_THRESHOLD was set to 10 and 50 are reported. In both cases, response-time mutations caused by decay of the prefetching code path are conservatively considered false positives, since these regressions were not the focus of this diagnosis effort.

8.5 Create behaviour

During development of Ursa Minor, we noticed that the distribution of request response times for CREATEs in postmark-large increased significantly during the course of the benchmark. To diagnose this performance degradation, we used Spectroscope to compare request flows between the first 1,000 CREATEs issued and the last 1,000. Due to the small number of requests compared, SM_THRESHOLD was set to 10.

Spectroscope's results showed categories that contained both structural and response-time mutations, with the highest-ranked one containing the former. The response-time mutations were the expected result of data structures in the NFS server and metadata server whose performance decreased linearly with load. Analysis of the structural mutations, however, revealed two architectural issues, which accounted for the degradation.

First, to serve a CREATE, the metadata server executed a tight inter-component loop with a storage node. Each iteration of the loop required a few milliseconds, greatly affecting response times. Second, categories containing structural mutations executed this loop more times than their precursor categories. This inter-component loop can be seen easily if the categories are zoomed out to show only component traversals and plotted in a train schedule, as in Figure 8.

Conversations with the metadata server's developer led us to the root cause: recursive B-Tree page splits needed to insert the new item's metadata. To ameliorate this problem, the developer increased the page size and changed the scheme used to pick the created item's key.

Summary: This problem demonstrates how request-flow comparison can be used to diagnose performance degradations, in this case due to a long-lived design problem. Though simple counters could have shown that CREATEs were very expensive, they would not have shown that the root cause was excessive metadata server/storage node interaction.

Quantitative analysis: For Table 3, results in the ranked list were deemed relevant if they contained structural mutations and showed more interactions between the NFS server and metadata server than their most-likely precursor category. Response-time mutations that showed expected performance differences due to load are considered false positives. Coverage is not reported as it is not clear how to define problematic CREATEs.

8.6 Slowdown due to code changes

This synthetic problem was injected into Ursa Minor to show how request-flow comparison can be used to diagnose slowdowns due to feature additions or regressions and to assess Spectroscope's sensitivity to changes in response time.

Spectroscope was used to compare request flows between two runs of SFS97. Problem period runs included a spin loop injected into the storage nodes' WRITE code path. Any WRITE request that accessed a storage node incurred this extra delay, which manifested in edges of the form $\star \rightarrow$ STORAGE_NODE_RPC_REPLY. Normally, these edges exhibit a latency of 100μs.

Table 3 shows results from injecting 100μs, 500μs, and 1ms spin loops. Results were deemed relevant if they contained response-time mutations and correctly identified the affected edges as those responsible. For the latter two cases, Spectroscope was able to identify the resulting response-time mutations and localize them to the affected edges. Of the categories identified, only 6–7% are false positives and 100% of the 10 highest-ranked ones are relevant. The coverage is 92% and 93%.

Variance in response times and the edge latencies in which the delay manifests prevent Spectroscope from properly identifying the affected categories for the 100μs case. It identifies 11 categories that contain requests that traverse the affected edges multiple times as containing

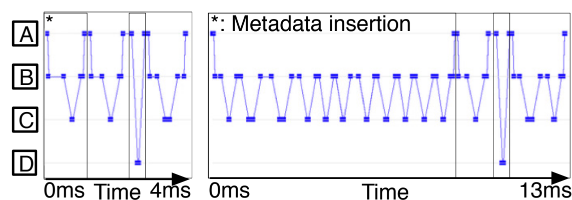


Figure 8: Visualization of create behaviour. Two train-schedule visualizations are shown, the first one a fast early create during postmark-large and the other a slower create issued later in the benchmark. Messages are exchanged between the NFS Server (A), Metadata Server (B), Metadata Storage Node (C), and Data Storage Node (D). The first phase of the create procedure is metadata insertion, which is shown to be responsible for the majority of the delay.

response-time mutations, but is unable to assign those edges as the ones responsible for the slowdown.

8.7 Periodic spikes

`Ursa minor-build`, which is run as part of the nightly test suite, periodically shows a spike in the time required for its copy phase to complete. For example, from one particular night to another, copy time increased from 111 seconds to 150 seconds, an increase of 35%. We initially suspected that the problem was due to an external process that periodically ran on the same machines as Ursa Minor's components. To verify this assumption, we compared request flows between a run in which the spike was observed and another in which it was not.

Surprisingly, Spectroscope's output contained only one result: GETATTRs, which were issued more frequently during the problem period, but which had not increased in average response time. We ruled this result out as the cause of the problem, as NFS's cache coherence policy suggests that an increase in the frequency of GETATTRs is the result of a performance change, not its cause. We probed the issue further by reducing `SM_THRESHOLD` to see if the problem was due to requests that had changed only a small amount in frequency, but greatly in response time, but did not find any such cases. Finally, to rule out the improbable case that the problem was caused by an increase in variance of response times that did not affect the mean, we compared distributions of intra-category variance between two periods using the Kolmogorov-Smirnov test; the resulting p-value was 0.72, so the null hypothesis was not rejected. These observations convinced us the problem was not due to Ursa Minor or processes running on its machines.

We next suspected the client machine as the cause of the problem and verified this to be the case by plotting a timeline of request arrivals and response times as seen by the NFS server (Figure 9). The visualization shows that during the problem period, response times stay constant but the arrival rate of requests decreases. We currently suspect the problem to be backup activity initiated from the facilities department (i.e., outside of our system).

Summary: This problem demonstrates how comparing request flows can help diagnose problems that are not caused by internal changes. Informing developers that nothing within the distributed system has changed frees them to focus their efforts on external factors.

9 Experiences at Google

This section describes preliminary experiences using request-flow comparison, as implemented in Spectroscope, to diagnose performance problems within select Google services. Sections 9.1 and 9.2 describe two such experiences. Section 9.3 discusses ongoing challenges in adapting request-flow comparison to large datacenters.

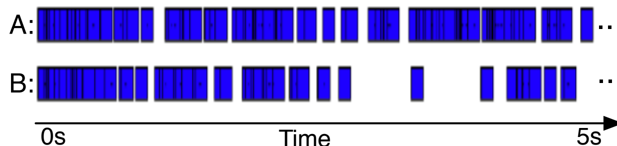


Figure 9: Timeline of inter-arrival times of requests at the NFS Server. A 5s sample of requests, where each rectangle represents the process time of a request, reveals long periods of inactivity due to lack of requests from the client during spiked copy times (B) compared to periods of normal activity (A).

9.1 Inter-cluster performance

A team responsible for an internal service at Google observed that load tests run on their software in two different clusters exhibited significantly different performance, though they expected performance to be similar.

We used Spectroscope to compare request flows between the two load test instances. The results showed many categories that contained response-time mutations; many were caused by latency changes not only within the service itself, but also within RPCs and within several dependencies, such as the shared Bigtable instance running in the lower-performing cluster. This led us to hypothesize that the primary cause of the slowdown was a problem in the cluster in which the slower load test was run. Later, we found out that the Bigtable instance running in the slower cluster was not working properly, confirming our hypothesis. This experience is a further example of how comparing request flows can help developers rule out the distributed system (in this case, a specific Google service) as the cause of the problem.

9.2 Performance change in a large service

To help identify performance problems, Google keeps per-day records of average request latencies for major services. Spectroscope was used to compare two day-long periods for one such service, which exhibited a significant performance deviation, but only a small difference in load, between the periods compared. Though many interesting mutations were identified, we were unable to identify the root cause due to our limited knowledge of the service, highlighting the importance of domain knowledge in interpreting Spectroscope's results.

9.3 Ongoing challenges with scale

Challenges remain in scaling request-flow comparison techniques to large distributed services, such as those within Google. For example, categories generated for well-instrumented large-scale distributed services will be much larger than those observed for the 5-instance version of Ursa Minor. Additionally, they may yield many categories, each populated with too few requests for sta-

tistical rigor. Robust methods are needed to merge categories and visualize them without losing important information about structure, which occurs with Dapper because of its graph compression methods. These methods affected the quality of Spectroscope's results by increasing variance, losing important structural differences between requests, and increasing effort needed to understand individual categories. Our experiences with unsupervised learning algorithms, such as clustering [4, 29], for merging categories indicate they are inadequate. A promising alternative is to use semi-supervised methods, which would allow the grouping algorithm to learn developers' mental models of which categories should be merged. Also, efficient visualization may be possible by only showing the portion of a mutation's structure that differs between it and its precursors.

More generally, request-flow graphs from large services are difficult to understand because such services contain many dependencies, most of which are foreign to their developers. To help, tools such as Spectroscope must strive to identify the semantic meaning of individual categories. For example, they could ask developers to name graph substructures about which they are knowledgeable and combine them into a meaningful meta-name when presenting categories.

10 Related work

A number of techniques have been developed for diagnosing performance problems in distributed systems. Whereas many rely on end-to-end tracing, others attempt to infer request flows from existing data sources, such as message send/receive events [27] or logs [38]. These latter techniques trade accuracy of re-constructed request flows for ease of using existing monitoring mechanisms. Other techniques rely on black-box metrics and are limited to localizing problems to individual machines.

Magpie [4], Pinpoint [9], WAP5 [27], and Xu [38], all identify anomalous requests by finding rare ones that differ greatly from others. In contrast, request-flow comparison identifies the changes in distribution between two periods that most affect performance. Pinpoint also describes other ways to use end-to-end traces, including for statistical regression testing, but does not describe how to use them to compare request flows.

Google has developed several analysis tools for use with Dapper [31]. Most relevant is the Service Inspector, which shows graphs of the unique call paths observed to a chosen function or component, along with the resulting call tree below it, allowing developers to understand the contexts in which the chosen item is used. Because the item must be chosen beforehand, the Service Inspector is not a good fit for problem localization tasks.

Pip [26] compares developer-provided, component-based expectations of structural and timing behaviour to actual behaviour observed in end-to-end traces. Theoretically, Pip can be used to diagnose any type of problem: anomalies, correctness problems, etc. But, it relies on developers to specify expectations, which is a daunting and error-prone task—the developer is faced with balancing effort and generality against the specificity needed to expose particular problems. In addition, Pip's component-centric expectations, as opposed to request-centric ones, complicate problem localization tasks [10]. Nonetheless, in many ways, comparing request flows between executions is akin to Pip, with developer-provided expectations being replaced with the observed non-problem period behaviour. Many of our algorithms, such as for ranking mutations and highlighting the differences, could be used with Pip-style expectations as well.

The Stardust tracing infrastructure on which our implementation builds was originally designed to enable performance models to be induced from observed system performance [32, 34]. Building on that initial work, IRONmodel [33] developed approaches to detecting (and correcting) violations of such models, which can indicate performance problems. In describing IRONmodel, Thereska et al. also proposed that the specific nature of how observed behaviour diverges from the model could guide diagnoses, but they did not develop techniques for doing so or explore the approach in depth.

A number of black-box diagnosis techniques have been devised for systems that do not have the detailed end-to-end tracing on which our approach to comparing request flows relies. For example, Project 5 [2] infers bottlenecks by observing messages passed between components. Comparison of performance metrics exhibited by systems that should be doing the same work can also identify misbehaving nodes [18, 24]. Such techniques can be useful parts of a suite, but are orthogonal to the contributions of this paper.

There are also many single-process diagnosis tools that inform creation of techniques for distributed systems. For example, Delta analysis [36] compares multiple failing and non-failing runs to identify the most significant differences. OptiScope [22] compares the code transformations made by different compilers to help developers identify important differences that affect performance. DARC [35] automatically profiles system calls to identify the greatest sources of latency. Our work builds on some concepts from such single-process techniques.

11 Conclusion

Comparing request flows, as captured by end-to-end traces, is a powerful new technique for diagnosing performance changes between two time periods or system versions. Spectroscope's algorithms for this comparison allow it to accurately identify and rank mutations and identify their precursors, focusing attention on the

most important differences. Experiences with Spectroscope confirm its usefulness and efficacy.

Acknowledgements

We thank our shepherd (Lakshminarayanan Subramanian), the NSDI reviewers, Brian McBarron, Michelle Mazurek, Matthew Wachs, and Ariela Krevat for their insight and feedback. We thank the members and companies of the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NEC Laboratories, NetApp, Oracle, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Yahoo! Labs) for their interest, insights, feedback, and support. This research was sponsored in part by a Google research award, NSF grants #CNS-0326453 and #CCF-0621508, by DoE award DE-FC02-06ER25767, and by CyLab under ARO grants DAAD19-02-1-0389 and W911NF-09-1-0273.

References

[1] M. Abd-El-Malek, et al. Ursa Minor: versatile cluster-based storage. Conference on File and Storage Technologies. USENIX Association, 2005. 2, 6

[2] M. K. Aguilera, et al. Performance debugging for distributed systems of black boxes. ACM Symposium on Operating System Principles. ACM, 2003. 13

[3] Anonymous. Personal communication with Google Software Engineers, December 2010. 1

[4] P. Barham, et al. Using Magpie for request extraction and workload modelling. Symposium on Operating Systems Design and Implementation. USENIX Association, 2004. 1, 2, 3, 13

[5] C. M. Bishop. *Pattern recognition and machine learning*, first edition. Springer Science + Business Media, LLC, 2006. 6

[6] B. M. Cantrill, et al. Dynamic instrumentation of production systems. USENIX Annual Technical Conference. USENIX Association, 2004. 1

[7] A. Chanda, et al. Whodunit: Transactional profiling for multi-tier applications. EuroSys. ACM, 2007. 1, 2

[8] F. Chang, et al. Bigtable: a distributed storage system for structured data. Symposium on Operating Systems Design and Implementation. USENIX Association, 2006. 8

[9] M. Y. Chen, et al. Path-based failure and evolution management. Symposium on Networked Systems Design and Implementation. USENIX Association, 2004. 1, 2, 13

[10] R. Fonseca, et al. Experiences with tracing causality in networked services. Internet Network Management Conference on Research on Enterprise Networking. USENIX Association, 2010. 2, 13

[11] R. Fonseca, et al. X-Trace: a pervasive network tracing framework. Symposium on Networked Systems Design and Implementation. USENIX Association, 2007. 1, 2

[12] GDB. http://www.gnu.org/software/gdb/. 1

[13] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003. 8

[14] S. L. Graham, et al. gprof: a call graph execution profiler. ACM SIGPLAN Symposium on Compiler Construction. Published as *SIGPLAN Notices*, **17**(6):120–126, June 1982. 1

[15] Graphviz. http://www.graphviz.org. 6

[16] J. Heer, et al. Prefuse: a toolkit for interactive information visualization. Conference on Human Factors in Computing Systems. ACM, 2005. 6

[17] 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. 10

[18] M. P. Kasick, et al. Black-box problem diagnosis in parallel file systems. Conference on File and Storage Technologies. USENIX Association, 2010. 13

[19] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997. 7

[20] F. J. Massey, Jr. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American Statistical Association*, **46**(253):66–78, 1951. 4

[21] J. C. Mogul. Emergent (Mis)behavior vs. Complex Software Systems. EuroSys. ACM, 2006. 1

[22] T. Moseley, et al. OptiScope: performance accountability for optimizing compilers. International Symposium on Code Generation and Optimization. IEEE/ACM, 2009. 13

[23] W. Norcott and D. Capps. IoZone filesystem benchmark program, 2002. http://www.iozone.org. 7

[24] X. Pan, et al. Ganesha: black-box fault diagnosis for MapReduce systems. Hot Metrics. ACM, 2009. 13

[25] J. R. Quinlan. Bagging, boosting and C4.5. 13th National Conference on Artificial Intelligence. AAAI Press, 1996. 6

[26] P. Reynolds, et al. Pip: Detecting the unexpected in distributed systems. Symposium on Networked Systems Design and Implementation. USENIX Association, 2006. 1, 13

[27] P. Reynolds, et al. WAP5: Black-box Performance Debugging for Wide-Area Systems. International World Wide Web Conference. ACM Press, 2006. 13

[28] R. R. Sambasivan, et al. *Diagnosing performance problems by visualizing and comparing system behaviours*. Technical report 10–103. Carnegie Mellon University, February 2010. 2

[29] R. R. Sambasivan, et al. Categorizing and differencing system behaviours. Workshop on hot topics in autonomic computing (HotAC). USENIX Association, 2007. 3, 13

[30] SPEC SFS97 (2.0). http://www.spec.org/sfs97. 2, 7

[31] B. H. Sigelman, et al. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical report dapper-2010-1. Google, April 2010. 1, 2, 13

[32] E. Thereska, et al. Informed data distribution selection in a self-predicting storage system. International conference on autonomic computing. IEEE, 2006. 13

[33] E. Thereska and G. R. Ganger. IRONModel: robust performance models in the wild. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008. 1, 13

[34] E. Thereska, et al. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2006. 1, 2, 13

[35] A. Traeger, et al. DARC: Dynamic analysis of root causes of latency distributions. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems. ACM, 2008. 13

[36] J. Tucek, et al. Triage: diagnosing production run failures at the user's site. ACM Symposium on Operating System Principles, 2007. 13

[37] E. R. Tufte. *The visual display of quantitative information*. Graphics Press, Cheshire, Connecticut, 1983. 3

[38] W. Xu, et al. Detecting large-scale system problems by mining console logs. ACM Symposium on Operating System Principles. ACM, 2009. 13

Profiling Network Performance for Multi-Tier Data Center Applications

*Minlan Yu** *Albert Greenberg†* *Dave Maltz†* *Jennifer Rexford** *Lihua Yuan†*
Srikanth Kandula† *Changhoon Kim†*
** Princeton University* *† Microsoft*

Abstract

Network performance problems are notoriously tricky to diagnose, and this is magnified when applications are often split into multiple tiers of application components spread across thousands of servers in a data center. Problems often arise in the communication between the tiers, where either the application or the network (or both!) could be to blame. In this paper, we present SNAP, a scalable network-application profiler that guides developers in identifying and fixing performance problems. SNAP passively collects TCP statistics and socket-call logs with low computation and storage overhead, and correlates across shared resources (e.g., host, link, switch) and connections to pinpoint the location of the problem (e.g., send buffer mismanagement, TCP/application conflicts, application-generated microbursts, or network congestion). Our one-week deployment of SNAP in a production data center (with over 8,000 servers and over 700 application components) has already helped developers uncover 15 major performance problems in application software, the network stack on the server, and the underlying network.

1 Introduction

Modern data-center applications, running over networks with unusually high bandwidth and low latency, should have great communication performance. Yet, these applications often experience low throughput and high delay *between* the front-end user-facing servers and the back-end servers that perform database, storage, and indexing operations. Troubleshooting network performance problems is hard. Existing solutions—like detailed application-level logs or fine-grain packet monitoring—are too expensive to run continuously, and still offer too little insight into where performance problems lie. Instead, we argue that data centers should perform continuous, lightweight profiling of the end-host network stack, coupled with algorithms for classifying and correlating performance problems.

1.1 Troubleshooting Network Performance

The nature of the data-center environment makes detecting and locating performance problems particularly challenging. Applications typically consist of tens to hundreds of application components, arranged in multiple tiers of front-ends and back-ends, and spread across hundreds to tens of thousands of servers. Application developers are continually updating their code to add features or fix bugs, so application components evolve independently and are updated while the application remains in operation. Human factors also enter into play: most developers do not have a deep understanding of how their design decisions interact with TCP or the network. There is a constant influx of new developers for whom the intricacies of Nagle's algorithm, delayed acknowledgments, and silly window syndrome remains a mystery.[1]

As a result, new network performance problems happen all the time. Compared to equipment failures that are relatively easy to detect, performance problems are tricky because they happen sporadically and many different components could be responsible. The developers sometimes blame "the network" for problems they cannot diagnose; in turn, the network operators blame the developers if the network shows no signs of equipment failures or persistent congestion. Often, they are both right, and the network stack or some subtle interaction between components is actually responsible [2, 3]. For example, an application sending small messages can trigger Nagle's algorithm in the TCP stack, causing transmission delays leading to terrible application throughput.

In the production data center we study, the process of actually detecting and locating even a single network per-

[1] Some applications (like memcached [1]) use UDP, and re-implement reliability, error detection, and flow control; however, these mechanisms can also introduce performance problems.

formance problem typically requires tens to hundreds of hours of the developers' time. They collect detailed application logs (too heavy-weight to run continuously), deploy dedicated packet sniffers (too expensive to run ubiquitously), or sample the data (too coarse-grained to catch performance problems). They then pore over these logs and traces using a combination of manual inspection and custom-crafted analysis tools to attempt to track down the issue. Often the investigation fails or runs out of time, and some performance problems persist for months before they are finally caught and corrected.

1.2 Lightweight, Continuous Profiling

In this paper, we argue that the data centers should *continuously* profile network performance, and analyze the data in real time to help pinpoint the source of the problems. Given the complexity of data-center applications, we cannot hope to fully automate the detection, diagnosis, and repair of network performance problems. Instead, our goal is dramatically reducing the demand for developer time by automatically identifying performance problems and narrowing them down to specific times and places (e.g., send buffer, delayed ACK, or network congestion). Any viable solution must be

- **Lightweight:** Running everywhere, all the time, requires a solution that is very lightweight (in terms of CPU, storage, and network overhead), so as not to degrade application performance.
- **Generic:** Given the constantly changing nature of the applications, our solution must detect problems without depending on detailed knowledge of the application or its log formats.
- **Precise:** To provide meaningful insights, the solution must pinpoint the component causing network performance problems, and tease apart interactions between the application and the network.

Finally, the system should help two very different kinds of users: (i) a *developer* who needs to detect, diagnose, and fix performance problems in his particular application and (ii) a *data-center operator* who needs to understand performance problems with the underlying platform so that he can tune the network stack, change server placement, or upgrade network equipment. In this paper, we present *SNAP (Scalable Network-Application Profiler)*, a tool that enables application developers and data-center operators to detect and diagnose these performance problems. SNAP represents an "existence proof" that a tool meeting our three requirements can be built, deployed in a production data center, and provide valuable information to both kinds of users.

SNAP capitalizes on the unique properties of modern data centers:

- SNAP has *full knowledge* of the network topology, the network-stack configuration, and the mapping of applications to servers. This allows SNAP to use correlation to identify applications with frequent problems, as well as congested resources (e.g., hosts or links) that affect multiple applications.
- SNAP can instrument the *network stack* to observe the evolution of TCP connections directly, rather than trying to infer TCP behavior from packet traces. In addition, SNAP can collect finer-grain information, compared to conventional SNMP statistics, without resorting to packet monitoring.

In addition, once the developers fix a problem (or the operator tunes the underlying platform), we can verify that the change truly did improve network performance.

1.3 SNAP Research Contributions

SNAP passively collects TCP statistics and socket-level logs in real time, classifies and correlates the data to pinpoint performance problems. The profiler quickly identifies the right location (end host, link, or switch), the right layer (application, network stack, or network), at the right time. Our major contributions of the paper are:

Efficient, systematic profiling of network-application interactions: SNAP provides a simple, efficient way to detect performance problems through real-time analysis of passively-collected measurements of the network stack. We provide a systematic way to identify the component (e.g., sender application, send buffer, network, or receiver) responsible for the performance problem. SNAP also correlates across connections that belong to the same application, or share underlying resources, to provide more insight into the sources of problems.

Performance characterization of a production data center: We deployed SNAP in a data center with over 8,000 servers, and over 700 application components (including map-reduce, storage, database, and search services). We characterize the sources of performance problems, which helps data-center operators improve the underlying platform and better tune the network.

Case studies of performance bugs detected by SNAP: SNAP pinpointed 15 significant and unexpected problems in application software, the network stack, and the interaction between the two. SNAP saved the developers significant effort in locating and fixing these problems, leading to large performance improvements.

Section 2 presents the design and development of SNAP. Section 3 describes our data-center environment

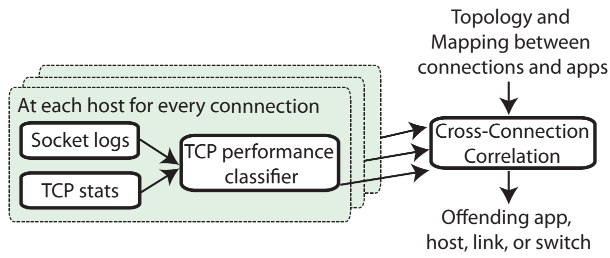


Figure 1: SNAP socket-level monitoring and analysis

and how SNAP was deployed. Section 4 validates SNAP against both labeled data (i.e., known performance problems) and detailed packet traces. Then, we present an evaluation of our one-week deployment of SNAP from the viewpoint of both the data-center operator (Section 5) and the application developer (Section 6). Section 7 shows how to reduce the overhead of SNAP through dynamic tuning of the polling rate. Section 8 discusses related work and Section 9 concludes the paper.

2 Design of the SNAP Profiler

In this section, we describe how SNAP pinpoints performance problems. Figure 1 shows the main components of our system. First, we collect *TCP-connection statistics*, augmented by *socket-level logs* of application read and write operations, in real time with low overhead. Second, we run a *TCP classifier* that identifies and categorizes periods of bad performance for each socket, and logs the diagnosis and a time sequence of the collected data. Finally, based on the logs, we have a *centralized correlator* that correlates across connections that share a common resource or belong to the same application to pinpoint the performance problems.

2.1 Socket-Level Monitoring of TCP

Data centers host a wide variety of applications that may use different communication methods and design patterns, so our techniques must be quite general in order to work across the space. The following three goals guided the design of our system, and led us *away* from using the SNMP statistics, packet traces, or application logs.

(i) Fine-grained profiling: The data should be fine-grained enough to indicate performance problems for individual applications on a small timescale (e.g, tens of milliseconds or seconds). Switches typically only capture link loads at a one-minute timescale, which is far too coarse-grained to detect many performance problems. For example, the TCP incast problem [3], caused by micro bursts of traffic at the timescale of tens of milliseconds, is not even visible in SNMP data.

Statistic	Definition
CurAppWQueue	# of bytes in the send buffer
MaxAppWQueue	Max # of bytes in send buffer over the entire socket lifetime
#FastRetrans	Total # of fast retransmissions
#Timeout	Total # of timeouts
#SampleRTT	Total # of RTT samples
#SumRTT	Sum of RTTs that TCP samples
RwinLimitTime	Cumulated time an application is receiver window limited
CwinLimitTime	Cumulated time an application is congestion window limited
SentBytes	Cumulated # of bytes the socket has sent over the entire lifetime
Cwin	Current congestion window
Rwin	Announced receiver window

Table 1: Key TCP-level statistics for each socket [5]

(ii) Low overhead: Data centers can be huge, with hundreds of thousands of hosts and tens of thousands sockets on each host. Yet, the data collection must not degrade application performance. Packet traces are too expensive to capture in real time, to process at line speed, or to store on disk. In addition, capturing packet traces on high-speed links (e.g., 1-10 Gbps in data centers) often leads to measurement errors including drops, additions, and resequencing of packets [4]. Thus it is impossible to capture packet trace everywhere, all the time to catch new performance problems.

(iii) Generic across applications: Individual applications often generate detailed logs, but these logs differ from one application to another. Instead, we focus on measurements that do not require application support so our tool can work across a variety of applications.

Through our work on SNAP, we found that the following two kinds of per-socket information can be collected cheaply enough to be used in analysis of large-scale data center applications, while still providing enough insight to diagnose where the performance problem lie (whether they are from the application software, from network issues, or from the interaction between the two).

TCP-level statistics: RFC 4898 [5] defines a mechanism for exposing the internal counters and variables of a TCP state-machine that is implemented in both Linux [6] and Windows [7]. We select and collect the statistics shown in Table 1 based on our diagnosis experience[2], which together expose the data-transfer performance of a socket. There are two types of statistics: (1) instantaneous snapshots (e.g., *Cwin*) that show the current value

[2]There are a few other variables in the TCP stack such as the time TCP spends in SlowStart stage, which are also useful but we did not mention in the paper due to space limit.

Locations	Problems	App/Net	Detection method
Sender app	Sender app limited	App	Not any other problems
Send buffer	Send buffer limited	App and Net	*CurAppWQueue* $\approx$ *MaxAppWQueue*
Network	Fast retransmission	Net	*diff(#FastRetrans)* > *0*
	Timeout	Net	*diff(#Timeout)* > *0)*
Receiver	Delayed ACK	App and Net	*diff(SumRTT)* > *diff(SampleRTT)*MaxQueuingDelay*
	Receiver window limited	App and Net	*diff(#RwinLimitTime)* > *0*

Table 2: Classes of network performance for a socket

of a variable in the TCP stack; and (2) cumulative counters (e.g., *#FastRetrans*) that count the number of events (e.g., the number of fast retransmissions) that happened over the lifetime of the socket. *#SampleRTT* and *SumRTT* are the cumulative values of the number of packets TCP sampled and the sum of the RTTs for these sampled packets. To calculate the retransmission timeout (RTO), TCP randomly samples one packet in each congestion window, and measures the time from the transmission of a packet to the time TCP receives the ACK for the packet as the RTT for this packet.

These statistics are updated by the TCP stack as individual packets are sent and received, making it too expensive to log every change of these values. Instead, we periodically poll these statistics. For the cumulative counters, we calculate the difference between two polls (e.g., *diff(#FastRetrans)*). For snapshot values, we sample with a Poisson interval. According to the PASTA property (Poisson Arrivals See Time Averages), the samples are a representative view of the state of the system.

Socket-call logs: Event-tracing systems in Windows [8] and Linux [9] record the time and number of bytes (*ReadBytes* and *WriteBytes*) whenever the socket makes a read/write call. Socket-call logs show the applications' data-transfer behavior, such as how many connections they initiated, how long they maintain each connection, and how much data they read/write (as opposed to the data that TCP actually transfers, i.e., *SentBytes*). These logs supplement the TCP statistics with application behavior to help developers diagnose problems. The socket-level logs are collected in an event-driven fashion, providing fine-grained information with low overhead. In comparison, the TCP statistics introduce a trade-off between accuracy and the polling overhead. For example, if SNAP polls TCP statistics once per second, a short burst of packet losses is hard to distinguish from a modest loss rate throughout the interval.

In summary, SNAP collects two types of data in the following formats: (i) timestamp, 4-tuples (source and destination address/port), *ReadBytes*, and *WriteBytes*; and (ii) timestamp, 4-tuples, TCP-level logs (Table 1). SNAP uses TCP-level logs to classify the performance problems and pinpoint the location of the problem, and then provides both the relevant TCP-level and socket-level logs for the affected connections for that period of time. Developers can use these logs to quickly find the root cause of performance problems.

2.2 Classifying Single-Socket Performance

Although it is difficult to determine the root cause of performance problems, we can pinpoint the component that is limiting performance. We classify performance problems in terms of the stages of data delivery, as summarized in the two columns of Table 2[3]:

1. Application generates the data: The sender application may not generate the data fast enough, either by design or because of bottlenecks elsewhere (e.g., CPU, memory, or disk). For example, the sender may write a small amount of data, triggering Nagle's algorithm [10] which combines small writes together into larger packets for better network utilization, at the expense of delay.

2. Data are copied from the application buffer to the send buffer: Even when the network is not congested, a small send buffer can limit throughput by stalling application writes. The send buffer must keep data until acknowledgments arrive from the receiver, limiting the buffer space available for writing new data.

3. TCP sends the data to the network: A congested network may drop packets, leading to lower throughput or higher delay. The sender can detect packet loss by receiving three duplicate ACKs, leading to a fast retransmission. When packet losses do not trigger a triple duplicate ACK, the sender must wait for a retransmission timeout (RTO) to detect loss and retransmit the data.

4. Receiver receives the data and sends an acknowledgment: The receiver may not read data, or acknowledge their arrival, quickly enough. The receiver window can limit the throughput if the receiver is not reading the data quickly enough (e.g., caused by a CPU starvation), allowing data to fill the receive buffer. A receiver delays sending acknowledgments in the hope of piggybacking the ACK on data in the reverse direction. The receiver acknowledges every other packet and waits up to 200 ms before sending an ACK.

[3]The table only summarizes major performance problems and can be extended to cover other problems such as out-of-order packets.

The TCP statistics provide direct visibility into certain performance problems like packet loss and receiver-window limits, where cumulative counts (e.g., *#Timeout*, *#FastRetrans*, and *RwinLimitTime*) indicate whether the problem occurred at any time during the polling interval. Detecting other problems relies on an instantaneous snapshot, such as comparing the current backlog of the send buffer (*CurAppWQueue*) to its maximum size (*MaxAppWQueue*); polling with a Poisson distribution allows SNAP to accurately estimate the fraction of time a connection is send-buffer limited. Pinpointing other latency problems requires some notion of expected delays. For example, the RTT should not be larger than the propagation delay plus the maximum queuing delay (*MaxQueuingDelay*) (whose value is measured in advance by operators), unless a problem like delayed ACK occurs. SNAP incorporates knowledge of the network configuration to identify these parameters.

SNAP detects send-buffer, network, and receiver problems using the rules listed in the last column of Table 2, where multiple problems may take place for the same socket during the same time interval. If any of these problems are detected, SNAP logs the diagnosis and all the variables in Table 1—as well as *WriteBytes* from the socket-call data—to provide the developers with detailed information to track down the problem. In the absence of any of the previous problems, we classify the connection as sender-application limited during the time interval, and log only the socket-call data to track application behavior. Being sender-application limited should be the most common scenario for a connection.

2.3 Correlation Across TCP Connections

Although SNAP can detect performance problems on individual connections in isolation, combining information across multiple connections helps pinpoint the location of the problem. As such, a central controller analyzes the results of the TCP performance classifier, as shown earlier in Figure 1. The central controller can associate each connection with a particular application and with shared resources like a host, links, and switches.

Pinpointing resource constraints (by correlating connections that share a host, link, or switch): Topology and routing data allow SNAP to identify which connections share resources such as a host, link, top-of-rack switch, or aggregator switch. SNAP checks if a performance problem (as identified by the algorithm in Table 2) occurs on many connections traversing the same resource at the same time. For example, packet losses (i.e., *diff(#FastRetrans)* > 0 or *diff(#Timeout)* > 0) on multiple connections traversing the same link would indicate a congested link. This would detect congestion occurring on a much smaller timescale than SNMP could measure. As another example, send-buffer problems for many connections on the same host could indicate that the machine has insufficient memory or a low default configuration of the send-buffer size.

Pinpoint application problem (by correlating across connections in the same application): SNAP also receives a mapping of each socket (as identified by the four-tuple) to an application. SNAP checks if a performance problem occurs on many connections from the same application, across different machines and different times. If so, the application software may not interact well with the underlying TCP layer. With SNAP, we have found several application programs that have severe performance problems and are currently working with developers to address them, as discussed in Section 6.

The two kinds of correlation analysis are similar, except for (i) sets of connections to compare S (i.e., connections sharing a resource vs. belonging to the same service) and (ii) the timescale for the comparison — correlation interval T (i.e., transient resource constraining events taking a few minutes or hours vs. permanent service code problems that lasts for days).

We use a simple linear correlation heuristic that works well in our setting Given a set of connections S and a correlation interval T, the SNAP correlation algorithm outputs whether these connections have correlated performance problems, and provides a time sequence of SNAP logs for operators and developers to diagnose.

We construct a performance vector $\overrightarrow{P_T(c,t)} = (time_k(p_1,c), ..., time_k(p_5,c))_{k=1..\lceil T/t \rceil}$, where t is an aggregation time interval in T and $time_k(p_i)(i = 1..5)$ denotes the total time that connection c is having problem p_i during time period $[(k-1)t, kt]$.[4] We pick c_1 and c_2 in S, calculate the Pearson correlation coefficient, and check if the average across all pairs of connections (*Average Correlation Coefficient ACC*) is larger than a threshold α:

$$ACC = \underset{c_1,c_2 \in S, c_1 \neq c_2}{avg} (cor(\overrightarrow{P_T(c_1,t)}, \overrightarrow{P_T(c_2,t)}) > \alpha,$$

where

$$cor(\overrightarrow{x}, \overrightarrow{y}) = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 (y_i - \bar{y})^2}}.$$

If the correlation coefficient is high, SNAP reports that the connections in S have a common problem. To extend this correlation for different classes of problems (e.g., one connection's delayed ACK problem triggers

[4] $p_i(i = 1..5)$ are the problems of send buffer limited, fast retransmission, timeout, delayed ACK and receiver window limited respectively. We do not include sender application limited because its time could be determined given the times of the first five problems.

Characteristic	Value
#Hosts	8K
#Applications	700
Operating systems	Win 2003,2008R2
Default send buffer	8 KB
Maximum segment size (MSS)	1460 Bytes
Minimum retrans. timeout	300 ms
Delayed ACK timeout	200 ms
Nagle's algorithm	mostly off
Slow start restart	off
Receiver window autotuning	off

Table 3: Characteristics in the production data center.

the sender application limited problem on another connection), we can extend our solution to use other inference techniques [11, 12] or principal component analysis (PCA) [13].

In practice, we must choose t carefully. With a large value of t, SNAP only compares the coarse-grained performance between connections; for example, if $t = T$, we only check if two connections have the same performance problem with the same percentage of time. With a small t, SNAP can detect fine-grained performance problems (e.g., two connections experiencing packet loss at almost the same time), but are susceptible to clock differences of the two machines and any differences in the polling rates for the two connections. The aggregation interval t should be large enough to mask the differences between the clocks and cannot be smaller than the least common multiple of the polling intervals of the connections.

3 Production Data Center Deployment

We deployed SNAP in a production data center. This section describes the characteristics of the data center and the configuration of SNAP, to set the stage for the following sections.

3.1 Data Center Environment

The data center consists of 8K hosts and runs 700 application components, with the configuration summarized in Table 3. The hosts run either Windows Server 2008 R2 or Windows Server 2003. The default send buffer size is 8K, and the maximum segment size is 1460 Bytes. The minimum retransmission timeout for packet loss is set to 300 ms, and the delayed-acknowledgment timeout is 200 ms. These values in Windows OS are configured for Internet traffic with long RTT.

While the OS enables Nagle's algorithm (which combines small writes into larger packets) by default, most delay-sensitive applications disable Nagle's algorithm using the *NO_DELAY* socket option.

Most applications in the data center use persistent connections to avoid establishing new TCP connections whenever they have data to transmit. Slow-start restart is disabled to reduce the delay arising when applications transfer a large amount of data after an idle period over a persistent connection.

Receiver-window autotuning—a feature in Windows Server 2008 that allows TCP to dynamically tune the receiver window size to maximize throughput—is disabled to avoid bugs in the TCP stack (e.g., [14]). Windows Server 2003 does not support this feature.

3.2 SNAP Configuration

We ran SNAP continuously for a week in August 2010. The polling interval for TCP statistics follows the Poisson distribution with an average inter-arrival time of 500 ms. We collected the socket-call logs for all the connections from and to the servers running SNAP. Over the week, we collected less than 1 GB on each host per day and the total is just terabytes of logs for the week. This is a very small amount of data compared to packet traces which take more than 180 GB per host per day at a 1 Gbps link, even if we just keep packet header information.

To identify the connections sharing the same switch, link, and application, we collect the information about the topology, routing, and the mapping between sockets and applications in the data center. We collect topology and routing information from the data center configuration files. To identify the mapping between the sockets and applications, we first run a script at each machine to identify the process that created each socket. We then map the processes to the application based on the configuration file for the application deployment.

To correlate performance problems across connections using the correlation algorithm we proposed in Section 2.3, we chose two seconds as the aggregation interval t to summarize the time on each performance problems to mask time difference between machines. To pinpoint transient resource constraints which usually last for minutes or hours, we chose one hour as the correlation interval T. To pinpoint problems from application code which usually last for days, we chose 24 hours as the correlation interval T. We chose the correlation threshold $\alpha = 0.4$.[5]

[5] It is hard to determine the threshold α in practice. Operators can choose the top n shared resources/application code to investigate their performance problems.

4 SNAP Validation

To validate the design of SNAP in Section 2 and evaluate whether SNAP can pinpoint the performance problems at the right place and time, we take two approaches: First, we inject a few known problems in our production data center and check if SNAP correctly catches these problems; Second, to validate the decision methods that use inference to determine the performance class in Table 2 rather than observing from TCP statistics directly, we compare SNAP results against packet traces.

4.1 Validation by Injecting Known Bugs

To validate SNAP, we injected a few known data-center networking problems and verified if SNAP correctly classifies those problems for each connection. Next, running our correlation algorithm on the SNAP logs of these labeled problems together with the other logs from the data center, SNAP correctly pinpointed *all* the labeled problems. For brevity, we first discuss two representative problems in detail and then show how SNAP pinpoints problematic host for each of them.

Problems in receive-window autotuning: We first injected a receiver-window autotuning problem: This problem happens when a Windows Server 2008 R2 machine initiates a TCP connection to a Windows Server 2003 machine with a SYN packet that requests the receiver window autotuning feature. But due to a bug in the TCP stack of the Windows Server 2003[6], the 2003 server does not parse the request for the receiver window autotuning feature correctly, and returns the SYN ACK packet with a wrong format. As a result, the 2008 server tuned its receiver window to four Bytes, leading to low throughput and long delay.

To inject this problem, we picked ten hosts running Windows 2008 in the data center and turn on their receiver window autotuning feature. Each of the ten hosts initiated TCP connections to a HTTP server running Windows 2003 to fetch 20 files of 5KB each from a host running Windows 2003.[7] It took the Windows 2003 server more than 5 seconds to transfer each 5KB file. SNAP correctly reported that all these connections are receiver window limited all the time, and SNAP logs showed that the announced receiver window size (*RWin*) is 4 Byte.

TCP incast: TCP incast [3] is a common performance problem in data centers. It happens when an *aggregator* distributes a request to a group of *workers*, and after processing the requests, these workers send the responses back at almost the same time. These responses together overflow the switch on the path and experience significant packet losses.

[6] This bug is later fixed with a patch, but some machines do not have the latest patch.

[7] We ran ten hosts to the same 2003 server to validate if the SNAP can correlate these connections and pinpoint the server.

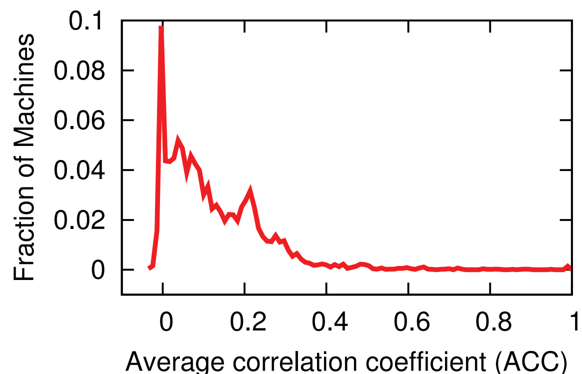


Figure 2: PDF of #Machines with different average correlation coefficient.

We wrote an application that generates a TCP incast traffic pattern. To limit the effect of our experiment to the other applications in the production data center, we picked 36 hosts under the same top-of-rack switch (TOR), used one host as the *aggregator* to send requests to the remaining 35 hosts which serve as *workers*. These workers respond with 100KB data immediately after they receive the requests. After receiving all the responses, the aggregator sends another request to the workers. The aggregator sends 20 requests in total.

SNAP correctly identified that seven of the 35 connections have experienced a significant amount of packet loss causing retransmission timeouts. This is verified from our application logs which show that it takes much longer time to get the response through the seven connections than the rest of the connections.[8]

Correlation to pinpoint resource constraints for the two problems: We mixed the SNAP logs of the receiver window autotuning problem and TCP incast with the logs of an hour period collected at all other machines in the data center. Then we ran SNAP correlation algorithm across the connections sharing the same machine.

SNAP correctly identified the Windows Server 2003 servers that have receiver-window limited problems across 5-10 connections with an average correlation coefficient (ACC) of 0.8. SNAP also correctly identified the aggregator machine because the ACC across all the connections that traverse the TOR is 0.45. Both are above

[8] In this experiment, SNAP can only tell that the connections have correlated timeouts. If the same problem happens for different aggregators running the same application code, we can tell that the application code causes the timeouts. If SNAP reports all the connections have simultaneous small writes (identified from socket call logs) and correlated timeouts, we can infer that the application code has incast problems.

the threshold $\alpha = 0.4$, which is chosen based on the discussion in Section 3.

Our correlation algorithm clearly distinguished the two injected problems with the performance of connections on the other machines in the data center. Figure 2 presents the probability density function (PDF) of the number of machines with different values of ACC. Only 2.7% of the machines have an ACC larger than 0.4. In addition to the two injected problems, the other machines with ACC > 0.4 may also indicate some problems that happen during our experiment, but we have not verified these problems yet.

4.2 Validation Against Packet Traces

We also need to validate the performance-classification algorithm defined in Table 2. The detection methods for the performance class of fast retransmissions, timeouts, receiver window limited is always accurate because these statistics are directly observed phenomena (e.g., #Timeouts) from the TCP stack. The accuracy of identifying send buffer problems is closely related to the probability of detecting the moments when the send buffer is full in the Poisson sampling, which is well studied in [15].

There is a tradeoff between the overhead and accuracy of identifying delayed ACK. The accuracy of identifying the delayed ACK and small writes classes is closely related to the estimation of the RTT. However, we cannot get per-packet RTT from the TCP stack because it is a significant overhead to log data for each packet. Instead, we get the sum of estimated RTTs (*SumRTT*) and the number of sampled packets (*SampleRTT*) from the TCP stack.

We evaluate the accuracy of identifying delayed ACK in SNAP by comparing SNAP's results with the packet trace. We picked two real-world applications from the production data center for which SNAP detects delayed ACK problems: One connection serves as an aggregator distributing requests for a Web application that has the delayed ACK problems for 100% of the packets[9]. Another belongs to a configuration-file distribution service for various jobs running in the data center, which has 75% of the packets on average experiencing delayed ACK. While running SNAP with various polling rates, we captured packet traces simultaneously. We then compared the results of SNAP with the number of delayed-ACK incidents we identify from packet traces.

To estimate the number of packets that experience delayed ACK, SNAP should find a distribution of RTTs for the sampled packets that sum up to *SumRTT*. Those packets that experience delayed ACK have a RTT around *DelayedACKTimeout*. The rest of the packets all experience the maximum queuing delay. Therefore, we use the equation: *(diff(#SumRTT) − diff(#SampleRTT) * MaxQueuingDelay)/DelayedACKTimeout* to count the number of packets experiencing delayed ACK. We use *MaxQueuingDelay* = 10 ms and *DelayedACKTimeout* = 180 ms. The delayed timeout is set as 200 ms in TCP stack, but TCP timer is only accurate at 10 ms level and thus the real *DelayedACKTimeout* varies around 200 ms. So we use 180 ms to be conservative on the delayed ACK estimation.

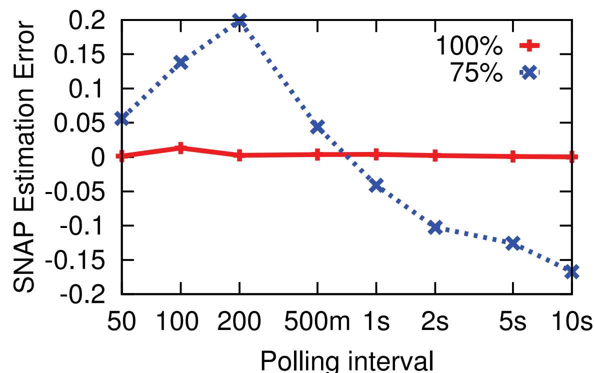


Figure 3: SNAP estimation error of identifying delayed ACK problems.

Figure 3 shows the estimation error of SNAP's results which is defined by $(d_t - d_s)/d_t$, where d_s is the percentage of packets that experience delayed ACK reported by SNAP and d_t is the actual percentage of delayed ACK we get from the packet trace. For the application that always has delayed ACK, SNAP's estimation error is 0.006 on average. For the application that has 75% of packets experiencing delayed ACK, the estimation error is within 0.2 for the polling intervals that range from 500 ms to 10 sec.

Figure 3 shows that the estimation error drops from positive (underestimation) to negative (overestimation) with the increase of the polling interval. When the polling interval is smaller than 200 ms, there is at most one packet experiencing delayed ACK in one polling interval. If a few packets take less than *MaxQueuingDelay* to transfer, we would overestimate the part of *SumRTT* that is contributed by these packets, and thus the rest of RTT is less than *DelayedACKTimeout*. When the polling interval is large, there are more packets experiencing delayed ACK in the same time interval. Since we have use 180 ms instead of 200 ms to detect delayed ACK, we would underestimate those packets that take longer than 180 ms delayed ACK. Nine such packets would contribute enough RTT for SNAP to assume one more delayed ACK.

[9]This application distributes requests whose size is smaller than MSS (i.e., one packet), and waits more than the delayed ACK timeout 200 ms before sending out another request. So the receiver has to keep each packet for 200 ms before sending the ACK to the sender.

5 Profiling Data Center Performance

We deployed SNAP in the production data center to characterize different classes of performance problems, and provided information to the data-center operators about problems with the network stack, network congestion or the interference between services. We first characterize the frequency of each performance problem in the data center, and then discuss the key performance problems in our data center—packet loss and the TCP send buffer.

5.1 Frequency of Performance Problems

Table 4 characterizes the frequency of the network performance problems (defined in Table 2) in our data center. Not surprisingly, the overall network performance of the data center is good. For example, only 0.82% of all the connections were receiver limited during their lifetimes. However, there are two key problems that the operators should address:

Operators should focus on the small fraction of applications suffering from significant performance problems. Several connections/applications have severe performance problems. For example, about 0.11% of the connections are receiver-window limited essentially all the time. Even though 0.11% sounds like a small number, when 8K machines are each running many connections, there is almost always some connection or application experiencing bad performance. These performance problems at the "tail" of the distribution also constrain the total load operators are willing to put in the data center. Operators should look at the SNAP logs of these connections and work with the developers to improve the performance of these connections so that they can safely "ramp up" the utilization of the data center.

Operators should disable delayed ACK, or significantly reduce Delayed ACK timeout: About two-thirds of the connections experienced delayed ACK problems. Nearly 2% of the connections suffer from delayed-ACKs for more than 99.9% of the time. We manually explore the delay-sensitive services, and count the percentage of connections that have delayed ACK. Unfortunately, about 136 delay-sensitive applications have experienced delayed ACKs. Packets that have delayed ACK would experience an unnecessary increase of latency by 200 ms, which is three orders of magnitude larger than the propagation delay in the data center and well exceeds the latency bounds for these applications. Since delayed ACK causes many problems for data-center applications, the operators are considering disabling delayed ACK or significantly reducing the delayed ACK timeout. The problems of delayed ACK for data center applications are also observed in [16].

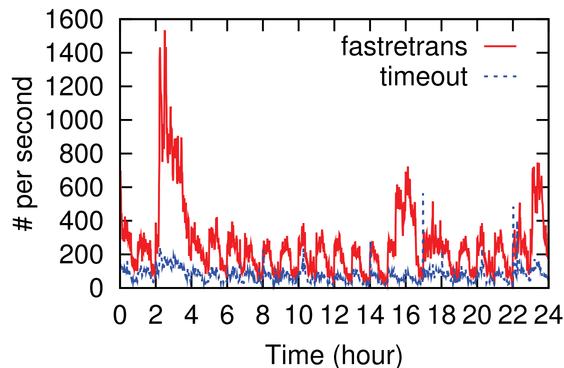


Figure 4: # of fast retransmissions and timeouts over time.

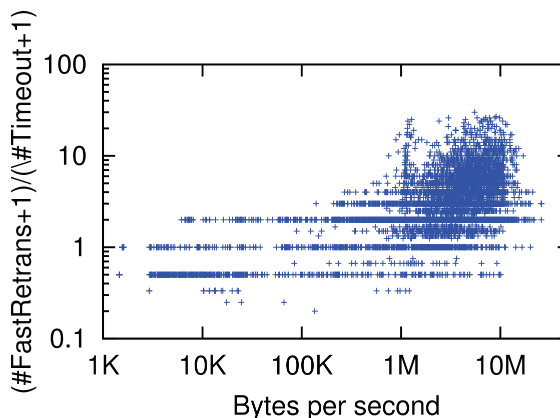


Figure 5: Comparing #FastRetrans and #Timeouts of flows with different throughput.

5.2 Packet Loss

Operators should schedule backup jobs more carefully to avoid triggering network congestion: Figure 4 shows the number of fast retransmissions and timeouts per second over time. The percentage of retransmitted bytes increases between 2 am and 4 am. This is because most backup applications with large bulk transfers are initiated in this time period.

Operators should reduce the number and effect of packet loss (especially timeouts) for low-rate flows: SNAP data shows that about 99.8% of the connections have low throughput ($<$ 1 MB/sec). Although these low-rate flows do not consume much bandwidth and are usually not the cause of network congestion, they are significantly affected by network congestion. Figure 5 is a scatter plot that shows the ratio of of fast retransmissions to timeouts vs. the connection sending rate. Each point in the graph represents one polling interval of one connection. Low-rate flows usually experience more timeouts than fast retransmission because they do not have multiple packets in flight to trigger triple duplicate ACKs. Timeouts, in turn, limit the throughput of these flows. In contrast, high-rate flows experience

Performance limitation	% of conn. with prob. for >X% of time >0	>25%	>50%	>75%	>99.9%	#Apps with prob. for >X% of time > 5%	> 50%
Sender app limited	97.91%	96.62%	89.61%	59.21%	32.61%	561	557
Send buffer limited	0.45%	0.06%	0.02%	0.01%	0.01%	1	1
Congestion	1.90%	0.46%	0.22%	0.17%	0.15%	30	6
Receiver window limited	0.82%	0.36%	0.21%	0.15%	0.11%	22	8
Delayed ACK	65.71%	33.20%	10.10%	3.21%	1.82%	154	144
(belong to delay sensitive apps)	63.52%	32.82%	9.71%	3.01%	1.61%	136	129

Table 4: Percentage of connections and number of applications that have different TCP performance limitations.

more fast retransmission than timeouts and can quickly recover from packet losses achieving higher throughput (> 1 MB/sec).

5.3 Send Buffer and Receiver Window

Operators should allow the TCP stack to automatically tune the send buffer and receiver window sizes, and consider the following two factors:

More send buffer problems on machines with more connections: SNAP reports correlated send buffer problems on hosts with more than 200 connections. This is because the larger the send buffer for each connection, the more memory is required for the machine. As a result, the developers of different applications on the same machine are cautious it setting the size of the send buffer; most use the default size of 8K, which is far less than the delay-bandwidth product in the data center and thus is more likely to become the performance bottleneck.

Mismatch between send buffer and receiver window size: SNAP logs the announced receiver window size when the connection is receiver limited. From the log we see that 0.1% of the total time when the senders indicate that their connections are bottlenecked by the receiver window, the receiver actually announced a 64 KB window. This is because the send buffer is larger than the announced receiver size, so the sender is still bottlenecked by the receiver window.

To fix the send-buffer problems in the short term, SNAP could help developers to decide what send buffer size they should set in an online fashion. SNAP logs the congestion window size (*CWin*), the amount of data the application expect to send (*WriteBytes*), and the announced receiver window size (*RWin*) for all the connections. Developers can use this information to size the send buffer based on the total resources (e.g., set the send buffer size to $Cwin_{thisconn} * TotalSendBufferMemory / \sum CWin$). They can also evaluate the effect of their change using SNAP. In the long term, operators should have the TCP stack automatically tune both the send-buffer and receiver-window sizes for all the connections (e.g., [6]).

6 Performance Problems Caught by SNAP

In this section, we show a few examples of performance problems caught by SNAP. In each example, we first show how the performance problem is exposed by SNAP's analysis of socket and TCP logs into performance classifications and then correlation across connections. Next, we explain how SNAP's reports help guide developers to identify quickly the root causes. Finally, we discuss the developer's fix or proposed fix to these problems. For most examples, we spent a few hours or days to discuss with developers to understand how their programs work and to discover how their programs cause the problems SNAP detects. It then took several days or weeks to iterate with developers and operators to find out the possible alternative ways to achieve their programing goals.

6.1 Sending Pattern/Packet Loss Issues

Spreading application writes over multiple connections lowers throughput: When correlating performance problems across connections from the same application, SNAP found one application whose connections always experienced more timeouts (*diff(#Timeout)*) than fast retransmission (*diff(#FastRetrans)*) especially when the *WriteBytes* is small. For example, SNAP reported repeated periods where one connection transferred an average of five requests per second with a size of 2 KB - 20 KB, while experiencing approximately ten timeouts but no fast retransmissions.

The developers were expecting to obtain far more than five requests per second from their system, and when this report showing small writes and timeouts was shown to them the cause became clear. The application sends requests to a server and waits for responses. Since some requests take longer to process than others and developers wanted to avoid having to implement request IDs while still avoiding head-of-line blocking, they open two connections to the server and place new requests on whichever connection is unused.

However, spreading the application writes over two connections meant that often there were not enough outstanding data on a connection to cause three duplicate

ACKs and trigger fast retransmission when a packet was lost. Instead, TCP fell back to its slower timeout mechanism.

To fix the problem, the application could send all requests over a single connection, give requests a unique ID, and use pools of worker threads at each end.[10] This would improve the chances there is enough data in flight to trigger fast retransmission when packet loss occurs.

Congestion window failing to prevent sudden bursts: SNAP discovered that some connections belonging to an application frequently experience packet loss (*#FastRetrans* and *#Timeout* are both high, and correlate strongly to the application and across time). SNAP's logs expose a time sequence of socket write logs (*WriteBytes*) and TCP statistics (*Cwin*) showing that before/during the intervals where packet loss occurs, there is a single large socket write call after an idle period. TCP immediately sends out the data in one large chunk of packets because the congestion window is large, but it experiences packet losses. For example, one application makes a socket call with *WriteBytes* > 100 *MB* after an idle period of 3 seconds, the *Cwin* is 64 KB, and the traffic burst leads to a bunch of packet losses.

The developers told us they use a persistent connection to avoid three-way handshake for each data transfer. Since "slow start restart" is disabled, the congestion window size does not age out and remains constant until there is a packet loss. As a result, the congestion window no longer indicates the carrying capacity of the network, and losses are likely when the application suddenly sends a congestion window worth of data.

Interestingly, the developers are opposed to enabling slow start restart, and they intentionally manipulate the congestion window in an attempt to reduce latency. For example, if they send 64 KB data, and the congestion window is small (e.g., 1 MSS), they need at multiple round-trip times to finish the data transfer. But if they keep the congestion window large, they can transfer the data with one RTT. In order to have a large congestion window, they first make a few small writes when they set up the persistent connection.

To reduce both the network congestion and delay, we need better scheduling of traffic across applications, allowing delay-sensitive applications to send traffic bursts when there is no network congestion, but pacing the traffic if the network is highly used. The feedback mechanism proposed in DCTCP [17] could be applied here.

Delayed ACK slows recovery after a retransmission timeout: SNAP found that one application frequently had two problems (timeout and delayed ACK) at almost the same time. As shown in Figure 6, when the fourth packet of the transferred data is lost, the TCP sender waits for a retransmission timeout (because there are not enough following packets to trigger triple-duplicate ACKs). However, the congestion window drops to one after the retransmission. As a result, TCP can only send a *single* packet, and the receiver waits for a delayed ACK timeout before acknowledging the packet. Meanwhile, the sender cannot increase its sending window until it receives the ACK from the receiver. To avoid this, developers are discussing the possibility of dropping the congestion window down to two packets when a retransmission timeout occurs. Disabling delayed ACK is another option.

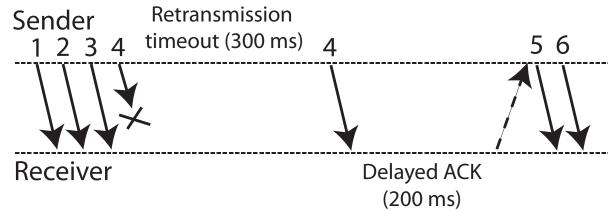


Figure 6: Delayed ACK after a retransmission timeout.

6.2 Buffer management and Delayed ACK

Some developers do not manage the application buffer and the socket send buffer appropriately, leading to bad interactions between buffer management and delayed ACK.

Delayed ACK caused by setting send buffer to zero: SNAP reports show that some applications have delayed ACK problems most of the time and these applications had set their send socket buffer length to 0. Investigation found that these applications set the size of the socket send buffer to zero in the expectation that it will decrease latency because data is not copied to a kernel socket buffer, but sent directly from the user space buffer. However, when send buffer is zero, the socket layer locks the application buffer until the data is ACK'd by the receiver so that the socket can retransmit the data in case a packet is lost. As a result, additional socket writes are blocked until the previous one has finished.

Whenever an application writes data that results in an odd number of packets being sent, the last packet is not ACK'd until the delayed ACK timer expires. This effectively blocks the sending application for 200 ms and can reduce application throughput to 5 writes per second. One team attempted to improve application performance by shrinking the size of their messages, but ended up creating an odd number of packets and triggering this issue — destroying the application's performance instead of helping it. After the developers increased the send buffer

[10]Note that the application should use a single connection because its requests are relatively small. For those applications that have a large amount of data to transfer for each request, they still have to use two connections to avoid head of line blocking during the network transfer.

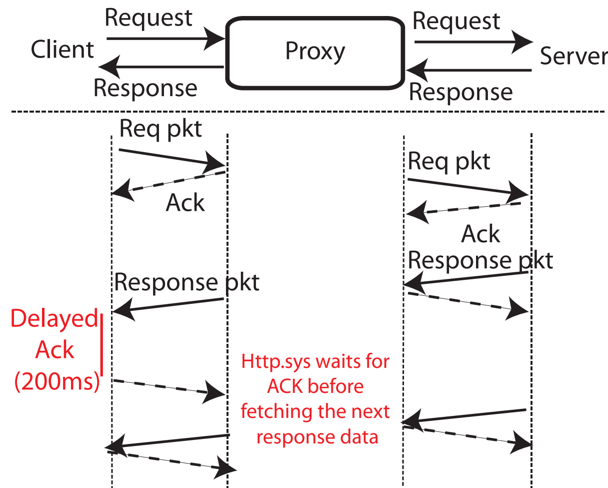


Figure 7: Performance problem in pipeline communication.

size, throughput returned to normal.

Delayed ACK affecting throughput: SNAP reports showed that an application was writing small amounts of data to the socket (*WriteBytes*) and its connections experienced both delayed ACK and sender application limited issues. For example, during 30 minutes, the application wrote 10K records at only five records per second and with a the record size of 20–100 Bytes.

The developers explained theirs is a logging application where the client uploads records to a server, and should be able generate far more than five records per second. Looking into the code with the developers, we found three key problems in the design: (i) *Blocking write:* to simplify the programming, the client does blocking writes and the server does blocking reads. (ii) *Small receive buffer:* The server calls recv() in a loop with a 200 bytes buffer in hopes that exactly one record is read in each receive call. (iii) *Send buffer is set to zero:* Since the application is delay-sensitive, the developer set send buffer size to zero. The application records are 20–100 Bytes — much less than the MSS of 1460 Bytes. Additionally, Nagle's algorithm forces the socket to wait for an ACK before it can send another packet (record).[11] As a result, the *single* packet containing each record always experience delayed ACK, leading to a throughput of only five records per second. To address this problem while still avoiding the buffer copying in memory, developers changed the sender code to write a group of requests each time. Throughput improved to 10K requests/sec after the change—a factor of 5000 improvement.

Delayed ACK affecting performance for pipelined applications: By correlating connections to the same machine, SNAP found two connections with performance problems that co-occur repeatedly: SNAP classified one connection as having a significant delayed ACK problem and the other as having sender application problems.

Developers told us that these two connections belong to the same application and form a pipeline pattern (Figure 7). There is a proxy that sits between the clients and servers and serves as a load balancer. The proxy passes requests from the client to the server, fetches a sequence of responses from the server, and passes them to the client. SNAP finds such a strong correlation between the delayed ACK problem and the receiver limited problem because both stem from the passing of the messages through the proxy.

After looking at the code, developers figured out that the proxy uses a single thread and a single buffer for both the client and the server. The proxy waits for the ACK of every transfer (one packet in each transfer most of the time) before fetching a new response data from the server.[12] When the developers changed the proxy to use two different threads with one fetching responses from the server and another sending responses to the client and a response queue between the two threads, the 99% tail of the request processing time drops from 200 ms to 10 ms.

6.3 Other Problems

SNAP has also detected other problems such as switch port failure (significant correlated packet losses across multiple connections sharing the same switch port), receiver window negotiation problems as reported in [14] (connections are always receiver window limited while receiver window size stays small), receiver not reading the data fast enough (receiver window limited), and poor latency caused by Nagle algorithm (sender application limited with small *WriteBytes*

7 Reducing SNAP CPU Overhead

To run in real time on all the hosts in the data center, SNAP must keep the CPU overhead and data volume low. The volume of data is small because (i) SNAP logs socket calls and TCP statistics instead of other high-overhead data such as packet traces and (ii) SNAP only logs the TCP statistics when there is a performance problem. To reduce CPU overhead, SNAP allows the operators to set the target percentage of CPU usage on each host. SNAP stays within a given CPU budget by dynamically tuning the polling rate for different connections.

[11] A similar performance problem caused by interactions between delayed ACK and Nagle is discussed in [10].

[12] The proxy is using the HTTP.sys library without setting the HTTP_SEND_RESPONSE_FLAG_BUFFER_DATA flag [18], which waits for the ACK from the client before sending a "send complete" signal to the application. By waiting for the ACK, HTTP.sys can make sure the application send buffer is not overwritten until the data is successfully transferred.

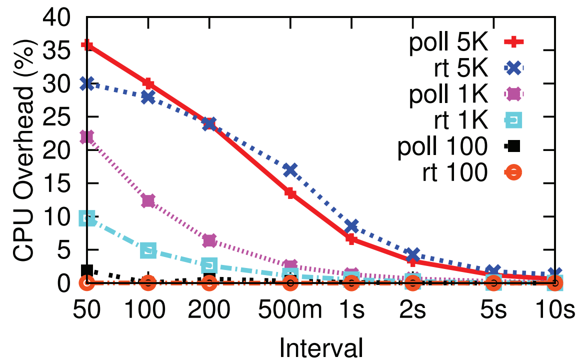


Figure 8: The CPU overhead of polling TCP statistics (*poll*) and reading TCP table (*rt*) with different number of connections (10, 100, 1K, 5K) and different intervals (from 50 ms to 10 sec).

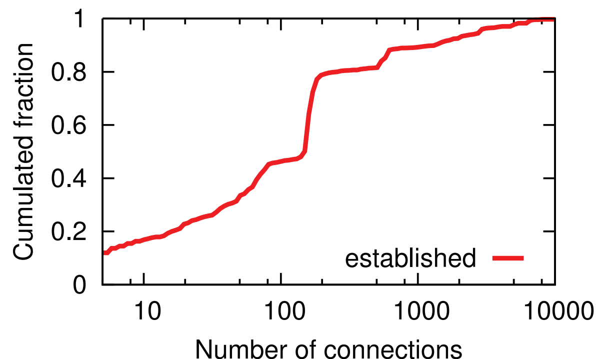


Figure 9: Number of connections per machine.

CPU Overhead of Profiling Since SNAP collects logs for all the connections at the host, the overhead of SNAP consists of three parts: logging socket calls, reading the TCP table, and polling TCP statistics.

Logging socket calls: In our data center, the cost of turning on the event tracing for socket logging is a median of 1.6% of CPU capacity [19].

Polling CPU statistics and reading TCP table: The CPU overhead of polling TCP statistics and reading the TCP table depends on the polling frequency and the number of connections on the machine. Figure 8 plots the CPU overhead on a 2.5 GHz Intel Xeon machine. If we poll TCP statistics for 1K connections at 500 millisecond interval, the CPU overhead is less than 5%. The CPU overhead of reading the TCP table is similar.

The CPU overhead is closely related to the number of connections on each machine. Figure 9 takes a snapshot of the distribution of the number of established connections per machine. There are at most 10K established sockets and a median of 150. This means operators can configure the interval of reading TCP table in most machines to be 500 millisecond or one second to keep the CPU overhead lower than 5%.[13] Since most of the connections in our data center are long-lived connections (e.g., persistent HTTP connections), we can read the TCP table at a lower frequency compared to TCP statistics polling. For the machines with many connections, we need to carefully adjust the polling rate of TCP statistics for each connection to achieve a tradeoff between diagnosis accuracy and the CPU overhead.

Dynamic Polling Rate Tuning To achieve the best tradeoff between CPU overhead and accuracy, operators can first configure l_{CPU} (u_{CPU}) to be the lower (upper) bound of the CPU percentage used by SNAP. We then propose an algorithm to dynamically tune the polling rate for different connections to keep CPU overhead between the two bounds. The basic idea of the algorithm is to have high polling rate for those connections that are having performance issues and have low polling rate for the others.

We start by polling all the connections on one host at the same rate. If the current CPU overhead is below l_{CPU}, we pick a connection that has the most performance problems in the past $T_{history}$ time, and increase its polling rate for more detailed data. Similarly if the current CPU overhead is above u_{CPU}, we pick a connection that has the least performance problems in the past $T_{history}$ time, and decrease its polling rate for more detailed data. Note that a lower polling rate introduces lower diagnosis accuracy. We can still catch those performance problems with the cumulative counters, but may miss some problems that rely on snapshots to detect.

8 Related Work

Previous work in diagnosing performance problems focuses on either the application layer or the network layer. SNAP addresses the interactions between them that cause particularly insidious performance issues.

In the application layer, prior work has taken several approaches: instrumenting application code [20, 21, 22] to find the causal path of problems, inferring the abnormal behaviors from history logs [11, 12], or identifying fingerprints of performance problems [23]. In contrast, SNAP focuses on profiling the interactions between applications and the network and diagnosing *network* performance problems, especially ones that arise from those interactions.

In the network layer, operators use network monitoring tools (e.g., switch counters) and active probing tools (ping, traceroute) to pinpoint network problems such as switch failures or congestion. To diagnose network performance problems, capture and analysis of packet traces remains the gold-standard. T-RAT [24] uses packet traces to diagnosis *throughput* bottlenecks in

[13] We read TCP tables at 500 millisecond interval in our data collection.

Internet traffic. Tcpanaly [4] uses packet traces to diagnose TCP stack problems. Others [25, 26] also infer the TCP performance and its problems from packet traces. In contrast, SNAP focuses on the multi-tier applications in data centers where it has access to the network stack, enabling us to create *simple* algorithms based on counters *far cheaper to collect than packet traces* to expose the network performance problems of the applications.

9 Conclusion

SNAP combines socket-call logs of the application's desired data-transfer behaviors with TCP statistics from the network stack that highlight the delivery of data. SNAP leverages the knowledge of topology, routing, and application deployment in the data center to correlate performance problems among connections, to pinpoint the congested resource or problematic software component.

Our experiences in the design, development, and deployment of SNAP demonstrate that it is practical to build a lightweight, generic profiling tool that runs continuously in the entire data center. Such a profiling tool can help both operators and developers in diagnosing network performance problems.

With applications in data centers getting more complex and more distributed, the challenges of diagnosing the performance problems between the applications and the network will only grow in importance in the years ahead. For future work, we hope to further automate the diagnosis process to save developers' efforts by exploring the appropriate variables to monitor in the stack, studying the dependencies between the variables SNAP collects, and combining SNAP reports with automatic analysis of application software.

Acknowledgments

We thank our shepherd Jason Flinn, the anonymous reviewers, Rob Harrison, Eric Keller, and Vytautas Valancius for their comments on earlier versions of this paper. We also thank Kevin Damour, Chuanxiong Guo, Randy Kern, Varugis Kurien, Saby Mahajan, Jitendra Padhye, Murari Sridharan, Ming Zhang for inspiring discussions on this paper.

References

[1] `http://memcached.org`.

[2] B. Krishnamurthy and J. Rexford, "HTTP/TCP Interaction," in *Web Protocols and Practice: HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement*, Addison-Wesley, 2001.

[3] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. Andersen, G. Ganger, G. Gibson, and B. Mueller, "Safe and effective fine-grained TCP retransmissions for datacenter communication," in *ACM SIGCOMM*, 2009.

[4] V. Paxson, "Automated packet trace analysis of TCP implementations," in *ACM SIGCOMM*, 1997.

[5] `http://www.ietf.org/rfc/rfc4898.txt`.

[6] `www.web100.org`.

[7] `http://msdn.microsoft.com/en-us/library/bb427395%28VS.85%29.aspx`.

[8] `http://msdn.microsoft.com/en-us/library/bb968803%28VS.85%29.aspx`.

[9] `http://datatracker.ietf.org/wg/syslog/charter/`.

[10] "TCP performance problems caused by interaction between Nagle's algorithm and delayed ACK." `www.stuartcheshire.org/papers/NagleDelayedAck`.

[11] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl, "Detailed diagnosis in computer networks," in *ACM SIGCOMM*, 2009.

[12] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *ACM SIGCOMM*, 2007.

[13] I. Jolliffe, *Principal Component Analysis*. Springer-Verlag, 1986.

[14] `support.microsoft.com/kb/983528`.

[15] C. Sarndal, B. Swensson, and J. Wretman, *Model Assisted Survey Sampling*. Springer-Verlag, 1992.

[16] A. Diwan and R. L. Sites, "Clock alignment for large distributed services," *Unpublished report*, 2011.

[17] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *ACM SIGCOMM*, 2010.

[18] `http://blogs.msdn.com/b/wndp/archive/2006/08/15/http-sys-buffering.aspx`.

[19] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of datacenter traffic: Measurements and analysis," in *Proc. Internet Measurement Conference*, 2009.

[20] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," in *NSDI*, 2004.

[21] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the unexpected in distributed systems," in *NSDI*, 2006.

[22] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-Trace: A pervasive network tracing framework," in *NSDI*, 2007.

[23] P. Bodik, M. Goldszmidt, A. Fox, D. B. Woodard, and H. Andersen, "Fingerprinting the datacenter: Automated classification of performance crises," in *EuroSys*, 2010.

[24] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker, "On the characteristics and origins of Internet flow rates," in *ACM SIGCOMM*, 2002.

[25] Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage, "Jigsaw: Solving the puzzle of enterprise 802.11 analysis," in *ACM SIGCOMM*, 2006.

[26] M. Tariq, A. Zeitoun, V. Valancius, N. Feamster, and M. Ammar, "Answering what-if deployment and configuration questions with WISE," in *ACM SIGCOMM*, 2008.

Efficiently Measuring Bandwidth at All Time Scales

Frank Uyeda* Luca Foschini† Fred Baker‡ Subhash Suri† George Varghese*
*U.C. San Diego †U.C. Santa Barbara ‡Cisco

Abstract

The need to identify correlated traffic bursts at various, and especially fine-grain, time scales has become pressing in modern data centers. The combination of Gigabit link speeds and small switch buffers have led to "microbursts", which cause packet drops and large increases in latency. Our paper describes the design and implementation of an efficient and flexible end-host bandwidth measurement tool that can identify such bursts in addition to providing a number of other features. Managers can query the tool for bandwidth measurements at resolutions chosen after the traffic was measured. The algorithmic challenge is to support such *a posteriori* queries without retaining the entire trace or keeping state for all time scales. We introduce two aggregation algorithms, Dynamic Bucket Merge (`DBM`) and Exponential Bucketing (`EXPB`). We show experimentally that `DBM` and `EXPB` implementations in the Linux kernel introduce minimal overhead on applications running at 10 Gbps, consume orders of magnitude less memory than event logging (hundreds of bytes per second versus Megabytes per second), but still provide good accuracy for bandwidth measures at any time scale. Our techniques can be implemented in routers and generalized to detect spikes in the usage of any resource at fine time scales.

1 Introduction

How can a manager of a computing resource detect bursts in resource usage that cause performance degradation without keeping a complete log? The problem is one of extracting a needle from a haystack; the problem gets worse as the needle gets smaller (as finer-grain bursts cause drops in performance) and the haystack gets bigger (as the consumption rate increases). While our paper addresses this general problem, we focus on detecting bursts of bandwidth usage, a problem that has received much attention [6, 16, 18] in modern data centers.

The simplest definition of a *microburst* is the transmission of more than B bytes of data in a time interval t on a single link, where t is in the order of 100's of microseconds. For input and output links of the same speed, bursts must occur on several links at the same time to overrun a switch buffer, as in the Incast problem [8, 16]. Thus, a more useful definition is the sending of more than B bytes in time t over *several* input links that are destined to the same output switch port. This general definition requires detecting bursts that are correlated in time across several input links.

Microbursts cause problems because data center link speeds have moved to 10 Gbps while commodity switch buffers use comparatively small amounts of memory (Mbytes). Since high-speed buffer memory contributes significantly to switch cost, commodity switches continue to provision shallow buffers, which are vulnerable to overflowing and dropping packets. Dropped packets lead to TCP retransmissions which can cause millisecond latency increases that are unacceptable in data centers.

Administrators of financial trading data centers, for instance, are concerned with the microburst phenomena [4] because even a latency advantage of 1 millisecond over the competition may translate to profit differentials of $100 million per year [14]. While financial networks are a niche application, high-performance computing is not. Expensive, special-purpose switching equipment used in high-performance computing (e.g. Infiniband and FiberChannel) is being replaced by commodity Ethernet switches. In order for Ethernet networks to compete, managers need to identify and address the fine-grained variations in latencies and losses caused by microbursts. At the core of this problem is the need to identify the bandwidth patterns and corresponding applications causing these latency spikes so that corrective action can be taken.

Efficient and effective monitoring becomes increasingly difficult as faster links allow very short-lived phenomenon to overwhelm buffers. For a commodity 24-port 10 Gbps switch with 4 MB of shared buffer, the buffer can be filled (assuming no draining) in 3.2 msec by a single link. However, given that bursts are often correlated across several links and buffers must be shared, the time scales at which interesting bursts occur can be ten times smaller, down to 100's of μs. Instead of 3.2 msec, the buffer can overflow in 320 μs if 10 input ports each receive 0.4 MB in parallel. Assume that the strategy to identify correlated bursts across links is to first identify bursts on single links and then to observe that they are correlated in time. The single link problem is then to efficiently identify periods of length t where more than B bytes of data occur. Currently, t can vary from hundreds of microseconds to milliseconds and B can vary from 100's of Kbytes to a few Mbytes. Solving this problem *efficiently* using minimal CPU processing and logging bandwidth is one of the main concerns of this paper.

Although identifying "bursts" on a single link for a range of possible time scales and byte thresholds is chal-

lenging, the ideal solution should do two more things. First, the solution should efficiently extract flows responsible for such bursts so that a manger can reschedule or rate limit them. Second, the tool should allow a manager to detect bursts correlated in time across links. While the first problem can be solved using heavy-hitter techniques [15], we briefly describe some new ideas for this problem in our context. The second problem can be solved by archiving bandwidth measurement records indexed by link and time to a relational database which can then be queried for persistent patterns. This requires an efficient summarization technique so that the archival storage required by the database is manageable.

Generalizing to Bandwidth Queries: Beyond identifying microbursts, we believe that modeling traffic at fine time scales is of fundamental importance. Such modeling could form the basis for provisioning NIC and switch buffers, and for load balancing and traffic engineering at fine time scales. While powerful, coarse-grain tools are available, the ability to flexibly and efficiently measure traffic at different, and especially fine-grain, resolutions is limited or non-existent.

For instance, we are unable to answer basic questions such as: what is the distribution of traffic bursts? At which time-scale did the traffic exhibit burstiness? With the identification of long-range dependence (LRD) in network traffic [9], the research community has undergone a mental shift from Poisson and memory-less processes to LRD and bursty processes. Despite its widespread use, however, LRD analysis is hindered by our inability to estimate its parameters unambiguously. Thus, our larger goal is to use fine-grain measurement techniques for fine-grain traffic modeling.

While it is not difficult to choose a small number of preset resolutions and perform measurements for those, the more difficult and useful problem is to support traffic measurements for *all time scales*. Not only do measurement resolutions of interest vary with time (as in burst detection), but in many applications they only become critical *after the fact*, that is, after the measurements have already been performed. Our paper describes an end-host bandwidth measurement tool that succinctly summarizes bandwidth information and yet answers general queries at arbitrary resolutions without maintaining state for all time scales.

Some representative queries (among many) that we wish such a tool to support are the following:

1. What is the *maximum* bandwidth used at time scale t?

2. What is the *standard deviation* and 95th percentile of the bandwidth at time scale t?

3. What is the *coarsest* time scale at which bandwidth exceeds threshold L?

In these queries, the query parameters t or L are chosen *a posteriori* — after all the measurements have been performed, and thus require supporting all possible resolutions and bandwidths.

Existing techniques: All the above queries above can be easily answered by keeping the entire packet trace. However, our data structures take an order of magnitude less storage than a packet trace (even a sampled packet trace) and yet can answer flexible queries with good accuracy. Note that standard summarization techniques (including simple ones like SNMP packet counters [1]) and more complex ones (e.g., heavy-hitter determination [13]) are very efficient in storage but must be targeted towards a particular purpose and at a fixed time scale. Hence, they cannot answer flexible queries for arbitrary time scales.

Note that sampling 1 in N packets, as in Cisco NetFlow [2], does not provide a good solution for bandwidth measurement queries. Consider a 10 Gbps link with an average packet size of 1000 bytes. This link can produce 10 million packets per second. Suppose the scheme does 1 in 1000 packet sampling. It can still produce 10,000 samples per second with say 6 bytes per sample for timestamp and packet size. To identify bursts of 1000 packets of 1500 bytes each (1.5 MB), any algorithm would look for intervals containing 1 packet and scale up by the down sampling factor of 1000. The major problem is that this causes false positives. If the trace is well-behaved and has no bursts in any specified period (say 10 msec), the scaling scheme will still falsely identify 1 in 1000 packets as being part of bursts because of the large scaling factor needed for data reduction. Packet sampling, fundamentally, takes no account of the passage of time.

From an information-theoretic sense, packet traces, are inefficient representations for bandwidth queries. Viewing a trace as a time series of point masses (bytes in each packet), it is more memory-efficient to represent the trace as a series of *time intervals* with bytes sent per interval. But this introduces the new problem of choosing the intervals for representation so that bandwidth queries on any interval (chosen after the trace has been summarized) can be answered with minimal error.

Our first scheme builds on the simple idea that for any fixed sampling interval, say 100 microseconds, one can easily compute traffic statistics such as $\max$ or *Standard Deviation* by a few counters each. By exponentially increasing the sampling interval, we can span an aggregation period of length T, and still compute statistics at all time scales from microseconds to milliseconds, using only $O(\log T)$ counters. We call this approach Exponential Bucketing (EXPB). The challenge in EXPB is to avoid updating all $\log T$ counters on each packet arrival and to prove error bounds.

Our second idea, dubbed Dynamic Bucket Merge

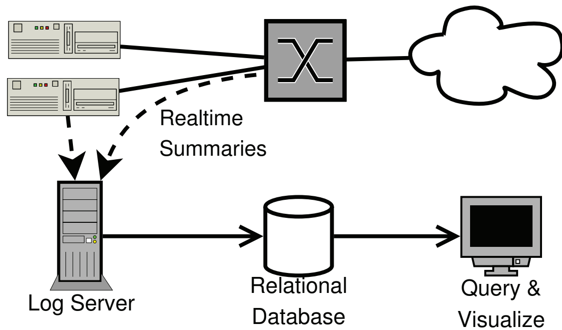


Figure 1: Example Deployment. End hosts and network devices implementing `EXPB` and `DBM` push output data over the network to a log server. Data at the server can be monitored and visualized by administrators then collapsed and archived to long-term, persistent storage.

(`DBM`), constructs an approximate streaming histogram of the traffic so that bursts stand out as peaks in this histogram. Specifically, we adaptively partition the traffic into k intervals/buckets, in such a way that the periods of heavy traffic map to more refined buckets than those of low traffic. The time-scales of these buckets provide a "visual history" of the burstiness of the traffic—the narrower the bucket in time, the burstier the traffic. In particular, `DBM` is well-suited for identifying not only whether a burst occurred, but *how many bursts*, and *when*.

System Deployment: Exponential Bucketing and Dynamic Bucket Merge have low computational and storage overheads, and can be implemented at multi-gigabit speeds in software or hardware. As shown in Figure 1, we envision a deployment scenario where both end hosts and network devices record fine-grain bandwidth summaries to a centralized log server. We argue that even archiving to a single commodity hard disk, administrators could pinpoint, to the second, the time at which correlated bursts occurred on given links, even up to a year after the fact.

This data can be indexed using a relational database, allowing administrators to query bandwidth statistics across links and time. For example, administrators could issue queries to "Find all bursts that occurred between 10 and 11 AM on all links in Set S". Set S could be the set of input links to a single switch (which can reveal Incast problems) or the path between two machines. Bandwidth for particular links can then be visualized to further delineate burst behavior. The foundation for answering such queries is the ability to efficiently and succinctly summarize the bandwidth usage of a trace in real-time, the topic of this paper.

We break down the remainder of our work as follows. We begin with a discussion of related algorithms and systems in Section 2. Section 3 illustrates the Dynamic Bucket Merge and Exponential Bucketing algorithms, both formally and with examples. We follow with our evaluations in Section 4, describe the implications for a system like Figure 1 in Section 5, and conclude in Section 6.

2 Related Work

Tcpdump [5] is a mature tool that captures a full log of packets at the endhost, which can be used for a wide variety of statistics, including bandwidth at any time scale. While flexible, tcpdump consumes too much memory for continuous monitoring at high speeds across every link and for periods of days. Netflow [2] can capture packet headers in routers but has the same issues. While sampled Netflow reduces storage, configurations with substantial memory savings cannot detect bursts without resulting in serious false positives. SNMP counters [1], on the other hand, provide packet and byte counts but can only return values at coarse and fixed time scales.

There are a wide variety of summarization data structures for traffic streams, many of which are surveyed in [15]. None of these can directly be adapted to solve the bandwidth problem at all time scales, though solutions to quantile detection do solve some aspects of the problem [15]. For example, classical heavy-hitters [13] measures the heaviest traffic flows during an interval. By contrast, we wish to measure "heavy-hitting subintervals across time", so to speak. However, heavy-hitter solutions are complementary in order to identify flows that cause the problem. The LDA data structure [12] is for a related problem – that of measuring average *latency*. LDA is useful for directly measuring latency violations. Our algorithms are complementary in that they help analyze the bandwidth patterns that *cause* latency violations.

`DBM` is inspired by the adaptive space partitioning scheme of [11], but is greatly simplified, and also considerably more efficient, due to the time-series nature of packet arrivals.

3 Algorithms

Suppose we wish to perform bandwidth measurements during a time window $[0, T]$, assuming, without loss of generality, that the window begins at time zero. We assume that during this period N packets are sent, with p_i being the byte size of the ith packet and t_i being the time at which this packet is logged by our monitoring system, for $i = 1, 2, \ldots, N$. These packets are received and processed by our system as a *stream*, meaning that the ith packet arrives before the jth packet, for any $i < j$.

The *bandwidth* is a rate, and so converting our observed sequence of N packets into a quantifiable bandwidth usage requires a *time scale*. Since we wish to measure bandwidth at different time scales, let us first make precise what we mean by this. Given a time

scale (or *granularity*) Δ, where $0 < \Delta < T$, we divide the measurement window $[0, T]$ into sub-intervals of length Δ, and aggregate all those packets that are sent within the same interval. In this way, we arrive at a sequence $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals.[1]

Therefore, every choice of Δ leads to a corresponding sequence S_Δ, which we interpret as the bandwidth use at the temporal granularity Δ. All statistical measurements of bandwidth usage at time scale Δ correspond to statistics over this sequence S_Δ. For instance, we can quantify the statistical behavior of the bandwidth at time scale Δ by measuring the *mean, standard deviation, maximum, median, quantiles*, etc. of S_Δ.

In the following, we describe two schemes that can estimate these statistics for *every a posteriori* choice of the time scale Δ. That is, after the packet stream has been processed by our algorithms, the users can *query* for an arbitrary granularity Δ and receive provable quality approximations of the statistics for the sequence S_Δ.

Our first scheme, `DBM`, is time scale agnostic, and essentially maintains a streaming histogram of the values $s_1, s_2, \ldots, s_k$, by adaptively partitioning the period $[0, T]$. Our second scheme `EXPB` explicitly computes statistics for *a priori* settings of Δ, and then uses them to approximate the statistics for the queried value of Δ.

Since the two schemes are quite orthogonal to each other, it is also possible to use them both in conjunction. We give worst-case error guarantees for both of the schemes. Both schemes are able to compute the mean with perfect accuracy and estimate the other statistics, such as the maximum or standard deviation, with a bounded error. The approximation error for the `DBM` scheme is expressed as an additive error, while the `EXPB` scheme offers a multiplicative relative error. In particular, for the `DBM` scheme, the estimation of the maximum or standard deviation is bounded by an error term of the form $O(\varepsilon B)$, where $0 < \varepsilon < 1$ is a user-specified parameter dependent on the memory used by the data structure, and $B = \sum_{i=1}^{N} p_i$ is the total packet mass over the measurement window. In the following, we describe and analyze the `DBM` scheme, followed by a description and analysis of the `EXPB` scheme.

3.1 Dynamic Bucket Merge

`DBM` maintains a partition of the measurement window $[0, T]$ into what we call *buckets*. In particular, a m-bucket partition $\{b_1, b_2, \ldots, b_m\}$, is specified by a sequence of time instants $t(b_i)$, with $0 < t(b_i) \le T$, with the interpretation that the bucket b_i spans the interval $(t(b_{i-1}),\ t(b_i)]$. That is, $t(b_i)$ marks the time when the ith bucket ends, with the convention that $t(b_0) = 0$, and $t(b_m) = T$. The number of buckets m is controlled by the memory available to the algorithm and, as we will show, the approximation quality of the algorithm improves linearly with m. In the following, our description and analysis of the scheme is expressed in terms of m. Each bucket maintains $O(1)$ information, typically the statistics we are interested in maintaining, such as the total number of bytes sent during the bucket. In particular, in the following, we use the notation $p(b)$ to denote the total number of data bytes sent during the interval spanned by a bucket b.

The algorithm processes the packet stream $p_1, p_2, \ldots, p_N$ in arrival time order, always maintaining a partition of $[0, T]$ into at most m buckets. (In fact, after the first m packets have been processed, the number of buckets will be exactly m, and the most recently processed packet lies in the last bucket, namely, b_m.) The basic algorithm is quite straightforward. When the next packet p_j is processed, we place it into a new bucket b_{m+1}, with time interval (t_{j-1}, T)—recall that t_{j-1} is the time stamp associated with the preceding packet p_{j-1}. We also note that the right boundary of the predecessor bucket b_m now becomes t_{j-1} due to the addition of the bucket b_{m+1}. Since we now have $m + 1$ buckets, we merge two *adjacent* buckets to reduce the bucket count down to m. Several different criteria can be used for deciding which buckets to merge, and we consider some alternatives later, but in our basic scheme we merge the buckets based on their *packet mass*. That is, we merge two adjacent buckets whose sum of the packet mass is the smallest over all such adjacent pairs. A pseudo-code description of `DBM` is presented in Algorithm 1.

Algorithm 1: `DBM`

```
1 foreach p_j ∈ S do
2 |   Allocate a new bucket b_i and set p(b_i) = p_j
3 |   if i == m + 1 then
5 |   |   Merge the two adjacent b_w, b_{w+1} for which
  |   |   p(b_w) + p(b_{w+1}) is minimum;
6 |   end
7 end
```

3.1.1 DBM Example

To clarify the operation of `DBM` we give the following example, illustrated in Figure 2.

Suppose that we run `DBM` with 4 buckets ($m = 4$), each of which stores a *count* of the number of buckets that have been merged into it, the *sum* of all bytes belonging to it, and the *max* number of bytes of any bucket merged into it. Now suppose that 4 packets have arrived

[1] To deal with the boundary problem properly, we assume that each sub-interval includes its right boundary, but not the left boundary. If we assume assume that no packet arrives at time 0, we can form a proper non-overlapping partition this way.

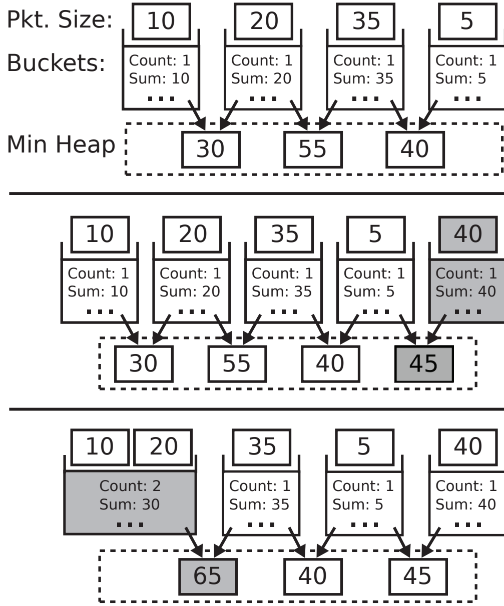


Figure 2: Dynamic Bucket Merge with 4 buckets. Initially each bucket contains a single packet and the min heap holds the sums of adjacent bucket pairs. When a new packet (value = 40) arrives, a 5th bucket is allocated and a new entry added to the heap. In the merge step, the smallest value (30) is popped from the heap and the two associated buckets are merged. Last, we update the heap values that depended on either of the merged buckets.

with masses 10, 20, 35, and 5, respectively. The state of DBM at this point is shown at the top of Figure 2. Note that Algorithm 1 required that we merge the buckets with the minimum combined sum. Hence, we maintain a min heap which stores the sums of adjacent buckets.

When a fifth packet with a mass of 40 arrives, DBM allocates a new bucket for it and updates the heap with the sum of the new bucket and its neighbor.

In the final step, the minimum sum is pulled from the heap and the buckets contributing to that sum are merged. In this example, the bucket containing mass 10 and 20 are merged into a single bucket with a new mass of 30 and a max bucket value of 20. Note that we also update the values in the heap which included the mass of either of the merge buckets.

3.1.2 DBM Analysis

The key property of DBM is that it can estimate the total number of bytes sent during *any* time interval. In particular, let $[t, t']$ be an arbitrary interval, where $0 \le t, t' \le T$, and let $p(t, t')$ be the total number of bytes sent during it, meaning $p(t,t') = \sum_{i=1}^{N} \{p_i \mid t \le t_i \le t'\}$. Then we have the following result.

Lemma 1. *The data structure* DBM *estimates* $p(t, t')$ *within an additive error* $O(B/m)$*, for any interval* $[t, t']$*, where* m *is the number of buckets used by* DBM *and* $B = \sum_{i=1}^{N} p_i$ *is the total packet mass over the measurement window* $[0, T]$.

Proof. We first note that in DBM each bucket's packet mass is at most $2B/(m-1)$, unless the bucket contains a single packet whose mass is strictly larger than $2B/(m-1)$. In particular, we argue that whenever two buckets need to be merged, there always exists an adjacent pair with total packet mass less than $2B/(m-1)$. Suppose not. Then, summing the sizes of all $(m-1)$ pairs of adjacent buckets must produce a total mass strictly larger than $2(m-1)B/(m-1) = 2B$, which is impossible since in this sum each bucket is counted at most twice, so the total mass must be less than $2B$.

With this fact established, the rest of the lemma follows easily. In order to estimate $p(t, t')$, we simply add up the buckets whose time spans intersect the interval $[t, t']$. Any bucket whose interval lies entirely inside $[t, t']$ is accurately counted, and so the only error of estimation comes from the two buckets whose intervals only partially intersect $[t, t']$—these are the buckets containing the endpoints t and t'. If these buckets have mass less than $2B/(m-1)$ each, then the total error in estimation is less than $4B/m$, which is $O(\frac{B}{m})$. If, on the other hand, either of the end buckets contains a single packet with large mass, then that packet is correctly included or excluded from the estimation, depending on its time stamp, and so there is no estimation error. This completes the proof. □

Theorem 1. *With* DBM *we can estimate the maximum or the standard deviation of* S_Δ *within an additive error* εB*, using memory* $O(1/\varepsilon)$.

Proof. The proof for the maximum follows easily from the preceding lemma. We simply query DBM for time windows of length Δ, namely, $(i\Delta, (i+1)\Delta]$, for $i = 0, 1, \ldots, \lceil T/\Delta \rceil$, and output the maximum packet mass estimated in any of those intervals. In order to achieve the target error bound, we use $m = \frac{4}{\varepsilon} + 1$ buckets.

We now analyze the approximation of the standard deviation. Recall that the sequence under consideration is $S_\Delta = \langle s_1, s_2, \ldots, s_k \rangle$, for some time scale Δ, where s_i is the sum of the bytes sent during the sub-interval $((i-1)\Delta, i\Delta]$, and $k = \lceil T/\Delta \rceil$ is the number of such intervals. Let $Var(S_\Delta)$, $E(S_\Delta)$, and $E(S_\Delta^2)$, respectively, denote the variance, mean, and mean of the squares for S_Δ. Then, by definition, we have

$$Var(S_\Delta) = E(S_\Delta^2) - E(S_\Delta)^2 = \frac{\sum_{i=1}^{k} {s_i}^2}{k} - E(S_\Delta)^2$$

Since DBM estimates each s_i within an additive error of εB, our estimated variance respects the following bound:

$$\leq \frac{\sum(s_i + \varepsilon B)^2}{k} - E(S_\Delta)^2$$

However, we can compute $E(S_\Delta)^2$ exactly, because it is just the square of the mean. In order to derive a bound on the error of the variance, we assume that $k > m$, that is, the size of the sequence S_Δ is at least as large as the number of buckets in DBM. (Naturally, statistical measurements are meaningless when the sample size becomes too small.) With this assumption, we have $2/k < 2/m$, and since $\varepsilon = 4/(m-1)$, we get that $2\frac{\sum s_i}{k} \leq \varepsilon B$, which, considering $k \geq 1$, yields the following upper bound for the estimated variance:

$$\leq \frac{\sum s_i^2}{k} - E(S_\Delta)^2 + \frac{k+1}{k}\varepsilon^2 B^2 \leq Var(S_\Delta) + 2\varepsilon^2 B^2$$

which implies the claim. □

Similarly, we can show the following result for approximating quantiles of the sequence S_Δ.

Theorem 2. *With* DBM *we can estimate any quantile of* S_Δ *within an additive error* εB*, using memory* $O(1/\varepsilon)$.

Proof. Let $s_1, s_2, \ldots, s_k$ be the sequence of data in the intervals $(i\Delta, (i+1)\Delta]$, for $i = 1, 2, \ldots, k = \lceil T/\Delta \rceil$, sorted in increasing order, and let $\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_k$ be the sorted estimated sequence for the same intervals. We now compute the desired quantile, for instance the 95th percentile, in this sequence. Supposing the index of the quantile is q, we return $\hat{s}_q$. We argue that the error of this approximation is $O(\varepsilon B)$. We do this by estimating bounds on the s_i values that are erroneously (due to approximation) misclassified, meaning reported below or equal the quantile when they are actually larger or vice versa. If no s_i have been misclassified then $\hat{s}_q$ and s_q correspond to the same sample, and by Lemma 1 the estimated value $\hat{s}_q - s_q \leq \varepsilon B$, hence the claim follows. On the other hand, if a misclassification occurred, then the sample s_q is reported at an index different than q in the estimated sequence. Assume without loss of generality that the sample s_q has been reported as $\hat{s}_u$ where $u > q$. Then, by the pigeonhole principle, there is at least a sample s_h ($h > q$) that is reported as $\hat{s}_d$, $d \leq q$. By Lemma 1, $\hat{s}_d - s_h \leq \varepsilon B$. Since s_q and s_h switched ranks in the estimated sequence $\hat{s}$, by Lemma 1 it holds that $s_h - s_q \leq \varepsilon B$ and $\hat{s}_u - \hat{s}_d \leq \varepsilon B$. By assumption $u > q \geq d$, then it follows that $\hat{s}_u \geq \hat{s}_q \geq \hat{s}_d$ in the sorted sequence $\hat{s}$, which implies that $\hat{s}_q - \hat{s}_d \leq \varepsilon B$. The chain of inequalities implies that $\hat{s}_q - s_q \leq 3\varepsilon B$, which completes the proof. □

Algorithm 1 can be implemented at the worst-case cost of $O(\log m)$ per packet, with the heap operation being the dominant step. The memory usage of DBM is $\Theta(m)$ as each bucket maintains $O(1)$ information.

3.1.3 Extensions to DBM for better burst detection

Generic DBM is a useful oracle for estimating bandwidth in any interval (chosen after the fact) with bounded additive error. However, one can tune the merge rule of DBM if the goal is to pick out the bursts only. Intuitively, if we have an aggregation period with k bursts for small k (say 10) spread out in a large interval, then ideally we would like to compress the large trace to k high-density intervals. Of course, we would like to also represent the comparatively low traffic adjacent intervals as well, so an ideal algorithm would partition the trace into $2k+1$ intervals where the bursts and ideal periods are clearly and even visually identified. We refer to the generic scheme discussed earlier that uses *merge-by-mass* as DBM-mm, and describe two new variants as follows.

- merge-by-variance (DBM-mv): merges the two adjacent buckets that have the minimum aggregated packet mass variance
- merge-by-range (DBM-mr): merges the two adjacent buckets that have the minimum aggregated packet mass range (defined as the difference between maximum and minimum packet masses within the bucket)

These merge variants can also be implemented in logarithmic time, and require storing $O(1)$ additional information for each bucket (in addition to $p(b_i)$).

One minor detail is that DBM-mv and DBM-mr are sensitive to null packet mass in an interval while DBM-mm is not. For these reasons, we make the DBM-mr and DBM-mv algorithms work on the sequence defined by S_Δ, where Δ is the minimum time scale at which bandwidth measurements can be queried. Then DBM-mr and DBM-mv represents S_Δ as a histogram on m buckets, where each bucket has a discrete value for the signal. The goal of a good approximation is to minimize its predicted value versus the true under some error metric. We consider both the L_2 norm and the L_∞ norm for the approximation error.

$$E_2 = \left(\sum_{i=1}^{n} |s_i - \hat{s}_i|^2\right)^{\frac{1}{2}} \quad (1)$$

where $\hat{s}_i$ is the approximation for value s_i.

$$E_\infty = \max_{i=1}^{n} |s_i - \hat{s}_i| \quad (2)$$

We compare the performance of DBM-mr and DBM-mv algorithms with the optimal offline algorithms, that is, a bucketing scheme that would find the optimal partition of S_Δ to minimize the E_2 or the E_∞ metric. Then, the analysis of [7, 10] can be adapted to yield the following results that formally state our intuitive goal of picking out m bursts with $2m+1$ pieces of memory.

Theorem 3. *The L_∞ approximation error of the m-bucket* `DBM-mr` *is never worse than the corresponding error of an optimal $m/2$-bucket partition.*

Theorem 4. *The L_2 approximation error of the m-bucket* `DBM-mv` *is at most $\sqrt{2}$ times the corresponding error of an optimal $m/4$-bucket partition.*

3.2 Exponential Bucketing

Our second scheme, which we call Exponential Bucketing (`EXPB`), explicitly computes statistics for *a priori* settings of $\Delta_1, \ldots, \Delta_m$, and then uses them to approximate the statistics for the queried value for *any* Δ, for $\Delta_1 \leq \Delta \leq \Delta_m$. We assume that the time scales grow in powers of two, meaning that $\Delta_i = 2^{i-1}\Delta_1$. Therefore, we can assume that the scheme processes data at the time scale Δ_1, namely, the sequence $S_{\Delta_1} = (s_1, s_2, \ldots, s_k)$.

Conceptually, `EXPB` maintains bandwidth statistics for all m time scales $\Delta_1, \ldots, \Delta_m$. A naïve implementation would require updating $O(m)$ counters per (aggregated) packet s_i. However, by carefully orchestrating the accumulator update when a new s_i is available it is possible to avoid spending m updates per measurement as shown in Algorithm 2.

The intuition is as follows. Suppose one is maintaining statistics at 100 μs and 200 μs intervals. When a packet arrives, we update the 100 μs counter but not the 200 μs counter. Instead, the 200 μs counter is updated only when the 100 μs counter is zeroed. In other words, only the lowest granularity counter is updated on every packet, and coarser granularity counters are only updated when all the finer granularity counters are zeroed.

Algorithm 2: `EXPB`

1 sum=$< 0, \ldots, 0 >$ (m times) ;
2 **foreach** s_i **do**
3   sum[0]=s_i;
4   j=0;
6   **repeat**
8   updatestat(j,sum[j]);
9   **if** $j < m$ **then**
10   sum[j+1]+=sum[j];
11   **end**
12   sum[j]=0;
13   j++;
14   **until** i *mod* $2^j \neq 0$ *or* $j \geq m$;
15 **end**

3.2.1 EXPB Example

To better understand the `EXPB` algorithm we now present the example illustrated in Figure 3.

In this example, we maintain 3 buckets ($m = 3$) each of which stores statistics at time scales of 1, 2 and 4

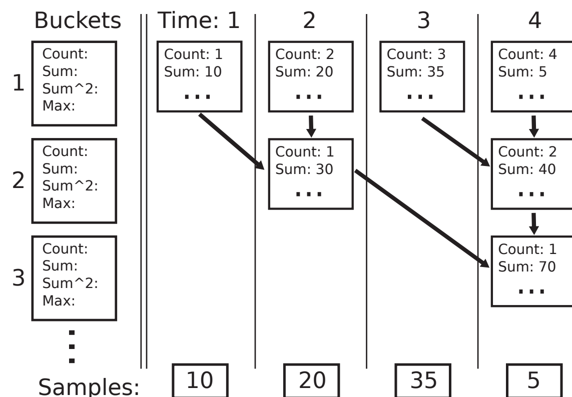


Figure 3: Exponential Bucketing Example. Each of the m buckets collects statistics at 2^{i-1} times the finest time scale. At the end of each time scale, Δ_i, buckets 1 to i must be updated. Before storing the new sum in a bucket j, we first add the old sum into bucket $j + 1$, if it exists.

time units. Each bucket stores the *count* of the intervals elapsed, the *sum* of the bytes seen in the current interval, and fields to compute max and standard deviation. We label the time units along the top and the number of bytes accumulated during each interval along the bottom.

In the first time interval 10 bytes are recorded in the first bucket and 10 is pushed to the sum of the second bucket. We repeat this operation when 20 is recorded in the second interval. Since 2 time units have elapsed, we also update the statistics for the Δ_2 time scale, and add bucket two's sum to bucket 3. In the third interval we update bucket 1 as before. Finally, at time 4 we update bucket 2 with the current sum from bucket 1, update bucket two's statistics, and push bucket two's sum to bucket 3. Finally, we update the statistics for Δ_3 with bucket three's sum.

3.2.2 EXPB Analysis

Algorithm 2 uses $O(m)$ memory and runs in $O(k)$ worst-case time, where $k = \lceil T/\Delta_1 \rceil$ is the number of intervals at the lowest time scale of the algorithm. The per-interval processing time is amortized constant, since the repeat loop starting at Line 6 simply counts the number of trailing zeros in the binary representation of i, for all $0 < i < k = T/\Delta$. The procedure $updatestat()$ called at Line 8 updates in constant time the $O(1)$ information necessary to maintain the statistics for each Δ_i, for $1 \leq i \leq m$.

We now describe and analyze the bandwidth estimation using `EXPB`. Given any query time scale Δ, we output the maximum of the bandwidth corresponding to the smallest index j for which $\Delta_j \geq \Delta$, and use the sum of squared packet masses stored for granularity Δ_j to compute the standard deviation. The following lemma bounds the error of such an approximation.

Lemma 2. *With* EXPB *we can return an estimation of the maximum or standard deviation of* S_Δ *that is between factor* $1/2$ *and* 3 *from the true value. The bound on the standard deviation holds in the limit when the ratio* $E(S_\Delta^2)/E(S_\Delta)^2$ *is large.*

Proof. We first prove the result for the statistic maximum, and then address the standard deviation. Let I be the interval $((i-1)\Delta, i\Delta]$ corresponding to the time scale Δ in which the maximum value is achieved, and let $p(I)$ be this value. Since $\Delta_j \geq \Delta$, there are at two most consecutive intervals I_i^j, I_{i+1}^j at time scale Δ_j that together cover I. By the pigeonhole principle, either I_i^j or I_{i+1}^j must contain at least half the mass of I, and therefore the maximum value at time scale Δ_j is at least 1/2 of the maximum value at Δ. This proves the lower bound side of the approximation. In order to obtain a corresponding upper bound, we simply observe that if I_i^j is the interval at time scale Δ_j with the maximum value, then I_i^j overlaps with at most 3 intervals of time scale Δ. Thus, the maximum value at time scale Δ_j cannot be more than 3 times the maximum at Δ proving an upper bound on the approximation.

The analysis for the standard deviation follows along the same lines, using the observation that $stddev_\Delta = \sqrt{E(S_\Delta^2) - E(S_\Delta)^2}$. An argument similar to the one used for the maximum value holds for the approximation of $E(S_\Delta^2)$. Then assuming the ratio $E(S_\Delta^2)/E(S_\Delta)^2$ to be a constant sufficiently greater than 1 implies the claim. We omit the simple algebra from this extended abstract. □

We note that there is a non-trivial extension of EXPB which allows it to work with a set of exponentially increasing time granularities whose common ratio can be any $\alpha > 1$. This can reduce average error. For a general $\alpha > 1$, Algorithm 2 cannot be easily adapted, so we need a generalization of it that uses an event queue while processing measurements to schedule when in the future a new measurement of length Δ_j must be sent to $updatestat()$. The details are omitted for lack of space.

3.3 Culprit Identification

As mentioned earlier, we do not want to simply identify bursts but also to *identify the flow* (e.g., TCP connection, or source IP address, protocol) that caused the burst so that the network manager can reschedule or move the offending station or application. The naive approach would be to add a heavy-hitters [13] data structure to each DBM bucket, which seems expensive in storage. Instead, we modify DBM to include two extra variables per bucket: a flowID and a flow count for the flowID.

The simple heuristic we suggest is as follows. Initially, each packet is placed in a bucket, and the bucket's flowID is set to the flowID of its packet. When merging two buckets, if the buckets have the same flowID, then that flowID becomes the flowID of the merged bucket and the flow counts are summed. If not, then one of the two flowIDs is picked with probability proportional to their flow counts. Intuitively, the higher count flows are more likely to be picked as the main contributor in each bucket as they are more likely to survive merges.

For EXPB, a simple idea is to use a standard heavy-hitters structure [13] corresponding to each of the logarithmic time scales. When each counter is reset, we update the flowID if the maximum value has changed and reinitialize the heavy-hitters structure for the next interval. This requires only a logarithmic number of heavy-hitters structures. Since there appears to be redundancy across the structures at each time scale, more compression appears feasible but we leave this for future work.

4 Evaluation

We now evaluate the performance and accuracy of DBM and EXPB to show that they fulfill our goal of a tool that efficiently utilizes memory and processing resources to faithfully capture and display key bandwidth measures. We will show that DBM and EXPB use significantly fewer resources than packet tracing and are suitable for network-wide measurement and visualization.

4.1 Measurement Accuracy

We implemented EXPB and the three variants of DBM as user-space programs and evaluated them with real traffic traces. Our traces consisted of a packets captured from the 1 Gigabit switch that connects several infrastructure servers used by the Systems and Networking group at U.C. San Diego, and socket-level send data produced by the record-breaking TritonSort sorting cluster [17].

Our "rsync" trace captured individual packets from an 11-hour period during which our NFS server ran its monthly backup to a remote machine using rsync. This trace recorded the transfer of 76.2 GB of data in 60.6 million packets, of which 66.6 GB was due to the backup operation. The average throughput was 15.4 Mbps with a maximum of 782 Mbps for a single second.

The "tritonsort" trace contains time-stamped byte counts from successful send system calls on a single host during the sorting of 500 GB of data using 23 nodes connected by a 10 Gbps network. This trace contains an average of 92,488 send events per second, with a peak of 123,322 events recorded in a single 1-second interval. In total, 20.8 GB were transferred over 34.24 seconds for an average throughput of 4.9 Gbps.

Ideally, our evaluation would include traffic from a mix of production applications running over a 10 Gbps network. While we do not have access to such a deployment, our traces provide insight into how DBM and EXPB might perform given the high bandwidth and network utilization of the "tritonsort" trace and the large variance in

bandwidth from second to second in the "rsync" trace.

For our accuracy evaluation, we used an aggregation period of 2 seconds. To avoid problems with incomplete sampling periods in EXPB, we must choose our time scales such that they all evenly divide our aggregation period. Since the prime factors of 2 seconds in nsec are 2^{11} and 5^{10} nsec, EXPB can use up to 11 buckets. Thus for EXPB, we choose the finest time scale to be $\Delta = 78.125$ μs (5^7 nsec) and the coarsest to be $\Delta =$ 80 msec ($2^{11}5^7$ nsec), which is consistent with the time scales for interesting bursts in data centers. For consistency, we also configure DBM to use a base sampling interval of 78.125 μs, but note that it can answer queries up to $\Delta = 2$ seconds.

To provide a baseline measurement, we computed bandwidth statistics for all of our traces at various time scales where $\Delta \geq 78.125$ μs. To ensure that all measurements in S_Δ are equal, we only evaluated time scales that evenly divided 2 seconds. In total, this provided us with ground-truth statistics at 52 different time scales ranging from 78.125 μs to 2 seconds. In the following sections we report accuracy in terms of error relative to these ground-truth measurements. While any number of values could be used for Δ and T in practice, we used these values across our experiments for the sake of a consistent and representative evaluation between algorithms.

4.1.1 Accuracy vs. Memory

We begin by investigating the tradeoff between memory and accuracy. At one extreme, SNMP can calculate average bandwidth using only a single counter. In contrast, packet tracing with tcpdump can calculate a wide range of statistics with perfect accuracy, but with storage cost scaling linearly with the number of packets. Both DBM and EXPB provide a tradeoff between these two extremes by supporting complex queries with bounded error, but with orders of magnitude less memory.

For comparison, consider the simplest event trace which captures a 64-bit timestamp and a 16-bit byte length for each packet sent or received. Using this data, one could calculate bandwidth statistics for the trace with perfect accuracy at a memory cost of 6 bytes *per event*. In contrast, DBM and EXPB require 8 and 16 bytes of storage per bucket used, respectively, along with a few bytes of meta data for each aggregation period.

To quantify these differences, we queried our traces for max, standard deviation, and 95th percentile (DBM only). For each statistic, we compute the average relative error of the measurements at each of our reference time scales and report the worst-case. To avoid spurious errors due to low sample counts, we omit accuracy data for standard deviations with fewer than 10 samples per aggregation period and 95th percentiles with fewer than 20 samples per aggregation period. We show the tradeoff between storage and accuracy in Table 1.

	Output	Max of Avg. Rel. Error		
		Max	S.Dev.	95th
trace (avg) (peak)	9.2 KBps 396 KBps	0%	0%	0%
DBM-mr	4 KBps	7.6%	14.7%	14.9%
EXPB	96 Bps	14.2%	5.9%	N/A

Table 2: We repeated our evaluation with the "rsync" trace and report accuracy results for our two best performing algorithms — DBM-mr and EXPB. We calculated the average relative error for each of our reference time scale and show the worst case.

While the simple packet trace gives perfectly accurate statistics, both DBM and EXPB consume memory at a fixed rate which can be configured by specifying the number of buckets and the aggregation period. In the presented configuration, both DBM and EXPB generate 4 KBps and 96 Bps, respectively — orders of magnitude less memory than the simple trace.

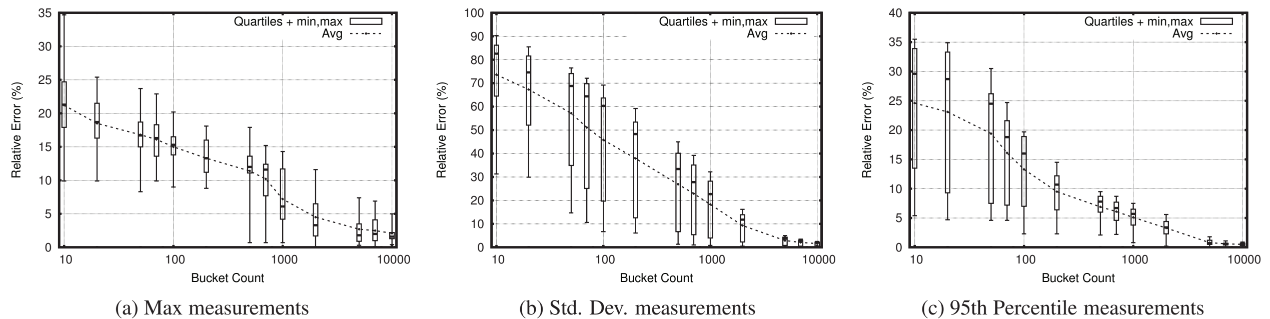
The cost of reduced storage overhead in DBM and EXPB is the error introduced in our measurements. However, we see that the range of average relative error rates is reasonable for max, standard deviation, and 95th percentile measurements. Further, of the DBM algorithms, DBM-mr gives the lowest errors throughout. While not shown, DBM's errors are largely due to under-estimation, but its accuracy improves as the query interval grows. EXPB gives consistent estimation errors for max across all of our reference points, but gradually degrades for standard deviation estimates as query intervals increase. Thus, for this trace, EXPB achieves the lowest error for query intervals less than 2msec. We have divided Table 1 to show the worst-case errors in these regions.

In Table 2 , we show the accuracy of DBM-mr and EXPB when run on the "rsync" trace with the same parameters as before. We note that again DBM-mr gives the most accurate results for larger query intervals, but now out-performs EXPB for query intervals greater than 160μs for max and 1msec for standard deviation.

To see the effect of scaling the number of buckets, we picked a representative query interval of 400 μs and investigated the accuracy of DBM-mr as the number of buckets were varied. The results of measuring the max, standard deviation and 95th percentile on the "tritonsort" trace are shown in Figure 4. We see that the relative error for all measurements decreases as the number of buckets is increased. However, at 4,000 buckets the curves flatten significantly and additional buckets beyond this do not produce any significant improvement in accuracy. While one might expect the error to drop to zero when the number of buckets is equal to the number of samples at S_Δ (5000 samples for 400μs), we do not see this since the trace is sampled at a finer granularity (78.125 μs) and

		Max of Avg. Relative Error					
		$\leq$ 2 msec			> 2msec		
	Output Rate	Max	S.Dev.	95th	Max	S.Dev.	95th
packet trace (avg)	555 KBps						
(peak)	740 KBps	0%	0%	0%	0%	0%	0%
DBM−mm, 1000 buckets	4 KBps	25.9%	43.3%	18.3%	2.2%	5.7%	1.1%
DBM−mv, 1000 buckets	4 KBps	16.7%	58.9%	26.7%	7.2%	39.0%	10.4%
DBM−mr, 1000 buckets	4 KBps	14.0%	35.0%	16.1%	2.0%	4.1%	0.9%
EXPB, 11 buckets	96 Bps	2.7%	2.5%	N/A	2.8%	8.1%	N/A

Table 1: Memory vs. Accuracy. We evaluate the "tritonsort" trace with a base time scale of Δ =78.125 μs and a 2 second aggregation period. Data output rate is reported for a simple packet trace compared with the DBM and EXPB algorithms. For each statistic, we compute the max of the average relative error of measurements for each of our reference time scales.



(a) Max measurements

(b) Std. Dev. measurements

(c) 95th Percentile measurements

Figure 4: Relative error for DBM−mr algorithm shown for the 400 μs time scale with a varying number of buckets. The box plots show the range of relative errors from the 25th to 75th percentiles, with the median indicated in between. The box whiskers indicate the min and max errors.

the buckets are merged online. There is no guarantee that DBM will merge the buckets such that each spans exactly 400μs of the trace.

With approximations of the max and standard deviation with this degree of accuracy, we see both DBM and EXPB as an excellent, low-overhead alternative to packet tracing.

4.1.2 DBM Visualization

One unique property of the DBM algorithms is that they can be visualized to show users the shape of the bandwidth curves. Note that we proved earlier that DBM−mr is optimal in some sense in picking out bursts. We now investigate experimentally how all DBM variants do in burst detection.

In Figures 5 we show the output for a single, 2 second aggregation period from the "rsync" trace using DBM−mr. For visual clarity, we configured DBM−mr to aggregate measurements at a 4 msec base time scale (250 data points) using 9 buckets. Figure 5 shows the raw data points (bandwidth use in each 4 msec interval of the 2 second trace) with the DBM−mr output superimposed. Notice that DBM−mr picks out four bursts (the vertical lines). The fourth burst looks smaller than the 3.1 Mbps burst observable in the raw trace. This is because there were two adjacent measurement intervals in the raw trace with bandwidths of 3.1 and 2.2 Mbps, respectively. DBM−mr merged these measurements into a single bucket of with an average bandwidth of 2.65 Mbps for 8 msec.

We show the output for all DBM algorithms in a more clean visual form in Figures 6a, 6b and 6c. We have normalized the width of the buckets and list their start and end times on the x-axis. Additionally, we label each bucket with its mass (byte count). This representation compresses periods of low traffic and highlights short-lived, high-bandwidth events. From the visualization of DBM−mr in Figure 6c, we can quickly see that there were four periods of time, each lasting between 4 and 8 msec where the bandwidth exceeded 2.3 Mbps. Note that in Figure 6a, DBM−mm picks out only two bursts. The remaining bursts have been merged into the three buckets spanning the period from 1440 to 1636 msec, thereby reducing the bandwidth (the y-axis) because the total time of the combined bucket increases.

In practice, a network administrator might want to

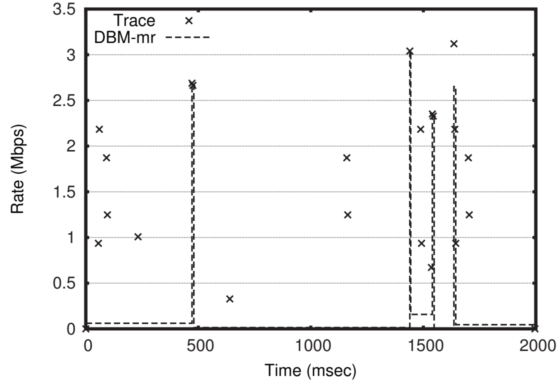


Figure 5: Visualization of events from a 2 second aggregation period overlaid with the output of DBM−mr using 9 buckets and a 4 msec measurement time scale.

quickly scan such a visualization and look for microburst events. To simulate such a scenario, we randomly inserted three bursts, each lasting 4 msec and transmitting between 4.0 and 4.4 MB of data. We show the DBM visualization for this augmented trace in bottom of Figure 6. DBM−mr and DBM−mm both allocate their memory resources to capture all three of these important events, even though they only represent 12 msec of a 2 second aggregation period. Again, DBM−mr cleanly picks out the three bursts.

4.1.3 Accuracy at High Load

As mentioned in Lemma 1, the error associated with the DBM algorithms increases with the ratio of total packet mass (total bytes) to number of buckets within an aggregation period. We now investigate to what extent increasing the mass within an aggregation period affects the measurement accuracy of DBM. To evaluate this, we first configured DBM to use a base time scale of $\Delta = 78.125\ \mu$s and 1000 buckets, as before, but vary the mass stored in DBM by changing the aggregation period. Figures 7a & 7b show the change in average relative error for both max and standard deviation statistics in our high-bandwidth "tritonsort" trace at a representative query time scale (400 μs) as the aggregation period is varied between 1 and 16 seconds.

For DBM−mm and DBM−mv with 1000 buckets the relative error diverges significantly as the aggregation period is increased. In contrast, DBM−mr shows only a subtle degradation for max from 5.9% to 12.3%. For standard deviation, DBM−mv show consistently poor performance with average relative errors increasing from 32% to 64%, while both DBM−mm and DBM−mr trend together with DBM−mr's errors ranging from 9.8 to 31.7%.

We contrast DBM−mr's performance for these experiments with that of EXPB. We see that EXPB's average relative error in the max measurement gradually falls from 2.8% to 1.9% as the aggregation period increases. Further, the error in standard deviation falls from 1.4% at a 1 second aggregation period to 0.5% at 16 seconds.

These results indicate that degradation in accuracy does occur as the ratio of the total packet mass to bucket count increases, as predicted by Lemma 1. While DBM must be configured correctly to bound the ratio of packet mass to bucket count, EXPB's accuracy is largely unaffected by the packet mass or aggregation period.

4.2 Performance Overhead

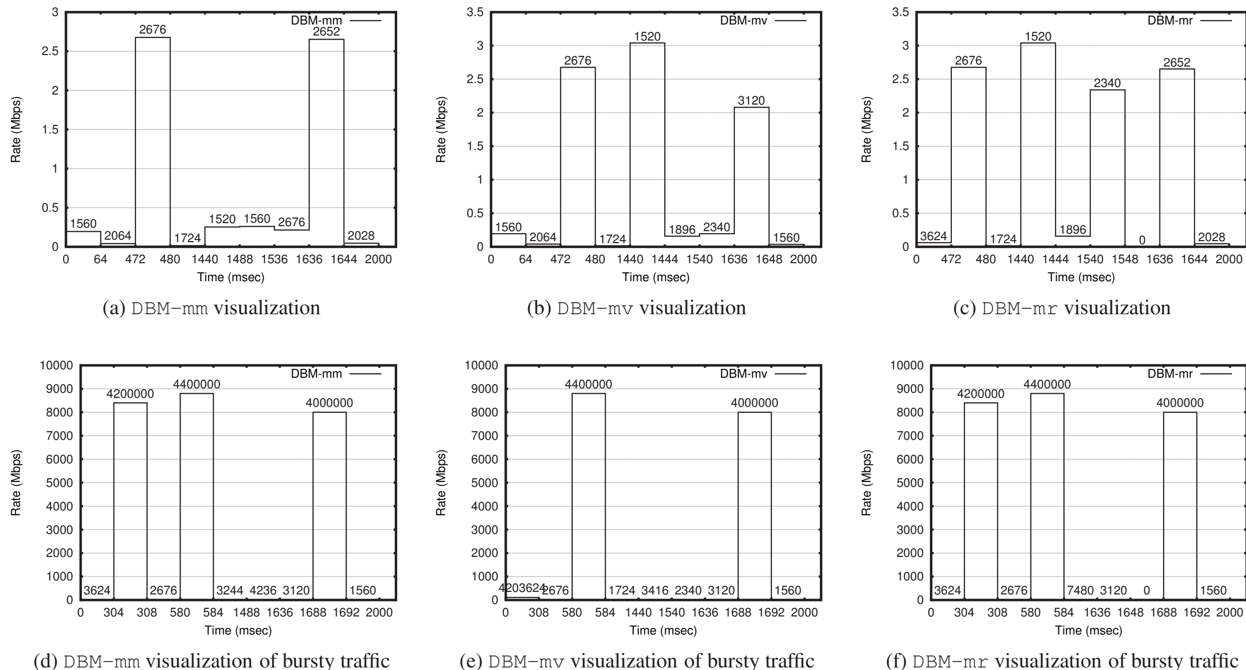
As previously stated, we seek to provide an efficient alternative to packet capture tools. Hence we compare the performance overhead of DBM and EXPB to that of an unmodified vanilla kernel, and to the well-established tcpdump[5].

We implemented our algorithms in the Linux 2.6.34 kernel along with a userspace program to read the captured statistics and write them to disk. To provide greater computational efficiency we constrained the base time scale and the aggregation period to be powers of 2. The following experiments were run on 2.27 GHz, quad-core Intel Xeon servers with 24 GB of memory. Each server is connected to a top-of-rack switch via 10 Gbps ethernet and has a round trip latency of approximately 100 μs.

To quantify the impact of our monitoring on performance, we first ran iperf [3] to send TCP traffic between two machines on our 10 Gbps network for 10 seconds. In addition, we instrumented our code to report the time spent in our routines during the test. We first ran the vanilla kernel source, then added different versions of our monitoring to aggregate 64 μs intervals over 1 second periods. We report both the bandwidth achieved by iperf and the average latency added to each packet at the sending server in Table 3. For comparison, we also report performance numbers for tcpdump when run with the default settings and writing the TCP and IP headers (52 bytes) of each packet directly to local disk. As DBM−mr is nearly identical to DBM−mm with respect to implementation, we omit DBM−mr's results.

As discussed in section 3.1, we see that the latency overhead per packet increases roughly as the log of the number of buckets. However, iperf's maximum throughput is not degraded by the latency added to each packet. Since the added latency per packet is several orders of magnitude less than the RTT, the overhead of DBM should not affect TCP's ability to quickly grow its congestion window. In contrast to DBM, tcpdump achieves 3.5% less throughput.

To observe the overhead of our monitoring on an application, we transferred a 1GB file using scp. We measured the wall-clock time necessary to complete the transfer by running scp within the Linux's time utility. To quantify the affects of our measurement on the total completion time, we measured the total overhead im-



(a) DBM−mm visualization

(b) DBM−mv visualization

(c) DBM−mr visualization

(d) DBM−mm visualization of bursty traffic

(e) DBM−mv visualization of bursty traffic

(f) DBM−mr visualization of bursty traffic

Figure 6: Visualization of DBM with 9 buckets over a single 2 second aggregation period. The start and end times for each bucket are shown on the x-axis, and each bucket is labeled with its mass (byte count). The top figures show the various DBM approximations of a single aggregation period, while the lower graphs show the same period with three short-lived, high bandwidth bursts randomly inserted.

posed on packets as they moved up and down the network stack. We report this overhead as a percentage of each experiment's average completion time (monitoring time divided by scp completion time). Each experiment was replicated 60 times and results are reported in Table 4. We see that although the cumulative overhead added by DBM grows logarithmically with the number of buckets, the time for scp to complete increases by at most 4.5%.

We see that our implementations of DBM and EXPB have a negligible impact on application performance, even while monitoring traffic at 10 Gbps.

4.3 Evaluation Summary

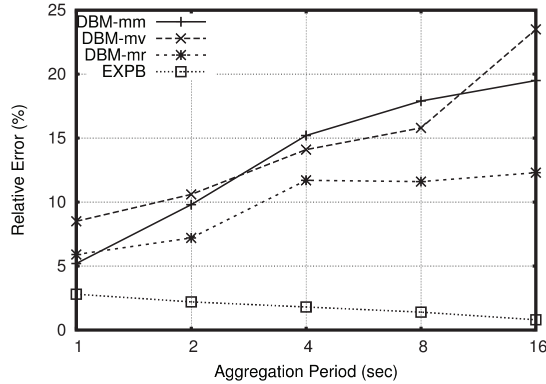
Our experiments indicate DBM−mr consistently provides better burst detection and has reasonable average case and worst case error for various statistics. When measuring at arbitrary time scales, EXPB has comparable or better average and worst-case error than DBM while using less memory. In addition, EXPB is unaffected by high mass in a given aggregation period. On other hand, DBM can approximate time series, which is useful for seeing how burst are distributed in time and for calculating more advanced statistics (i.e. percentiles). We recommend a parallel implementation where EXPB is used for Max and Standard Deviation and DBM−mr is used for all other queries.

5 System Implications

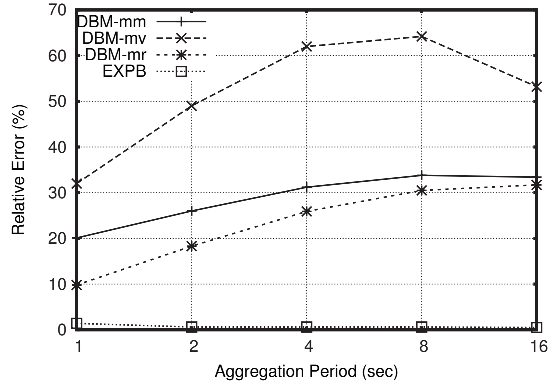
So far, we have described DBM and EXPB as part of an end host monitoring tool that can aggregate and visualize bandwidth data with good accuracy. However, we see these algorithms a part of a larger infrastructure monitoring system.

Long-term Archival and Database Support It is useful for administrators to retrospectively troubleshoot problems that are reported by customers days after the fact. At slightly more than 4 KBps, the data produced by both DBM and EXPB for a week (2.4 GB per link) could easily be stored to a commodity disk. With this data, an administrator can pinpoint traffic abnormalities at microsecond timescales and look for patterns across links. The data can be compacted for larger time scales by reducing granularity for older data. For example, one hour of EXPB data could be collapsed into one set of buckets containing max and standard deviation information at the original resolutions but aggregated across the hour.

With such techniques, fine-grain network statistics for hundreds of links over an entire year could be stored to a single server. The data could be keyed by link and time and stored in a relational database to allow queries across time (is the traffic on a single link becoming more bursty with time?) or across links (did a number of bursts cor-



(a) Max measurements with 1000 buckets



(b) Std. Dev. measurements with 1000 buckets

Figure 7: Average relative error for the DBM with 1000 buckets and EXPB with 11 buckets shown on the "tritonsort" trace for a 400 μs query interval and various aggregation periods.

Version	Buckets	Avg. BW	Overhead/Pkt
vanilla	N/A	9.053 Gbps	0.0 nsec
DBM-mm	10	9.057 Gbps	256.5 nsec
	100	9.010 Gbps	335.7 nsec
	1000	9.104 Gbps	237.5 nsec
	10000	8.970 Gbps	560.4 nsec
DBM-mv	10	9.043 Gbps	205.7 nsec
	100	8.986 Gbps	327.9 nsec
	1000	9.067 Gbps	432.2 nsec
	10000	9.067 Gbps	457.2 nsec
EXPB	14	9.109 Gbps	169.4 nsec
tcpdump	N/A	8.732 Gbps	N/A

Table 3: Average TCP bandwidth reported by iperf over 60 10-second runs. We also show the average time spent in the kernel-level monitoring functions for each packet sent. DBM and EXPB were run with a base time scale of $\Delta = 64$ μs and $T =$ 1 second aggregation period.

Version	Buckets	Time	Overhead
vanilla	N/A	14.133 sec	N/A
DBM-mm	10	14.334 sec	1.3%
	100	14.765 sec	1.9%
	1000	14.483 sec	2.7%
	10000	14.527 sec	2.5%
DBM-mv	10	14.320 sec	1.7%
	100	14.344 sec	2.3%
	1000	14.645 sec	2.9%
	10000	14.482 sec	3.1%
EXPB	14	14.230 sec	0.4%
tcpdump	N/A	15.253 sec	7.9%

Table 4: The time needed to transfer a 1GB file over scp. We measured the cumulative overhead incurred by our monitoring routines for all send and receive events. We report this overhead as a percentage of each experiment's total running time.

relate on multiple switch input ports?).

Hardware Implementation Both DBM and EXPB algorithms can be implemented in hardware for use in switches and routers. EXPB has an amortized cost of two bucket updates per measurement interval. Since bucket updates are only needed at the frequency of the measurement time scale, these operations could be put on a work queue and serviced asynchronously from the main packet pipeline. The key complication for implementing DBM in hardware is maintaining a binary heap. However, a 1000 bucket heap can be maintained in hardware using a 2-level radix-32 heap that uses 32-way comparators at 10 Gbps. Higher bucket sizes and speeds will require pipelining the heap. The extra hardware overhead for these algorithms in gates is minimal. Finally, the logging overhead is very small, especially when compared to NetFlow.

6 Conclusions

Picking out bursts in a large amount of resource usage data is a fundamental problem and applies to all resources, whether power, cooling, bandwidth, memory, CPU, or even financial markets. However, in the domain of data center networks, the increase of network speeds beyond 1 Gigabit per second and the decrease of in-network buffering has made the problem one of great interest.

Managers today have little information about how microbursts are caused. In some cases they have identified paradigms such as InCast, but managers need better visibility into bandwidth usage and the perpetrators of

microbursts. They would also like better understanding of the temporal dynamics of such bursts. For instance, do they happen occasionally or often? Do bursts linger below a tipping point for a long period or do they arise suddenly like tsunamis? Further, correlated bursts across links lead to packet drops. A database of bandwidth information from across an administrative domain would be valuable in identifying such patterns. Of course, this could be done by logging a record for every packet, but this is too expensive to contemplate today.

Our paper provides the first step to realizing such a vision for a cheap network-wide bandwidth usage database by showing efficient summarization techniques at links ($\sim$4 KB per second, for example, for running `DBM` and `EXPB` on 10 Gbps links) that can feed a database backend as shown in Figure 1. Ideally, this can be supplemented by algorithms that also identify the flows responsible for bursts and techniques to join information across multiple links to detect offending applications and their timing. Of the two algorithms we introduce, Exponential Bucketing offers accurate measurement of the average, max and standard deviation of bandwidths at arbitrary sampling resolutions with very low memory. In contrast, Dynamic Bucket Merge approximates a time-series of bandwidth measurements that can visualized or used to compute advanced statistics, such as quantiles.

While we have shown the application of `DBM` and `EXPB` to bandwidth measurements in endhosts, these algorithms could be easily ported to in-network monitoring devices or switches. Further, these algorithms can be generally applied to any time-series data, and will be particularly useful in environments where resource spikes must be detected at fine time scales but logging throughput and archival memory is constrained.

7 Acknowledgements

This research was supported by grants from the NSF (CNS-0964395) and Cisco. We would also like to thank our shepherd, Dave Maltz, and our anonymous reviewers for their insightful comments and feedback.

References

[1] A Simple Network Management Protocol (SNMP). http://www.ietf.org/rfc/rfc1157.txt.

[2] Cisco Netflow. www.cisco.com/web/go/netflow.

[3] iperf. http://http://iperf.sourceforge.net/.

[4] Performance Management for Latency-Intolerant Financial Trading Networks. *Financial Service Technology*, 9.

[5] tcpdump. http://www.tcpdump.org/.

[6] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). *SIGCOMM*, 2010.

[7] C. Buragohain, N. Shrivastava, and S. Suri. Space Efficient Streaming Algorithms for the Maximum Error Histogram. In *ICDE*, 2007.

[8] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP incast throughput collapse in datacenter networks. In *WREN 2009*.

[9] A. Erramilli, O. Narayan, and W. Willinger. Experimental queueing analysis with long-range dependent packet traffic. *IEEE/ACM Trans. Netw.*, 4(2):209–223, 1996.

[10] S. Gandhi, L. Foschini, and S. Suri. Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order. In *ICDE*, 2010.

[11] J. Hershberger, N. Shrivastava, S. Suri, and C. Toth. Adaptive Spatial Partitioning for Multidimensional Data Streams. *Algorithmica*, 2006.

[12] R. R. Kompella, K. Levchenko, A. C. Snoeren, and G. Varghese. Every microsecond counts: tracking fine-grain latencies with a lossy difference aggregator. In *SIGCOMM 2009*.

[13] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A low cost device for identifying large flows. In *HOTI*, 2007.

[14] R. Martin. Wall Street's Quest To Process Data At The Speed Of Light. *Information Week*, April 23 2007.

[15] S. Muthukrishnan. *Data streams: Algorithms and applications*. now Publishers Inc., 2005.

[16] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *FAST 2008*.

[17] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI 2011*.

[18] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. *SIGCOMM 2009*.

ETTM: A Scalable Fault Tolerant Network Manager

Colin Dixon Hardeep Uppal Vjekoslav Brajkovic Dane Brandon
Thomas Anderson Arvind Krishnamurthy
University of Washington

Abstract

In this paper, we design, implement, and evaluate a new scalable and fault tolerant network manager, called ETTM, for securely and efficiently managing network resources at a packet granularity. Our aim is to provide network administrators a greater degree of control over network behavior at lower cost, and network users a greater degree of performance, reliability, and flexibility, than existing solutions. In our system, network resources are managed via software running in trusted execution environments on participating end-hosts. Although the software is physically running on end-hosts, it is logically controlled centrally by the network administrator. Our approach leverages the trend to open management interfaces on network switches as well as trusted computing hardware and multicores at end-hosts. We show that functionality that seemingly must be implemented inside the network, such as network address translation and priority allocation of access link bandwidth, can be simply and efficiently implemented in our system.

1 Introduction

In this paper, we propose, implement, and evaluate a new approach to the design of a scalable, fault tolerant network manager. Our target is enterprise-scale networks with common administrative control over most of the hardware on the network, but with complex quality of service and security requirements. For these networks, we provide a uniform administrative and programming interface to control network traffic at a packet granularity, implemented efficiently by exploiting trends in PC and network switch hardware design. Our aim is to provide network administrators a greater degree of control over network behavior at lower cost, and network users a greater degree of performance, reliability, and flexibility, compared to existing solutions.

Network management today is a difficult and complex endeavor. Although IP, Ethernet and 802.11 are widely available standards, most network administrators need more control over network behavior than those protocols provide, in terms of security configuration [21, 14], resource isolation and prioritization [36], performance and cost optimization [4], mobility support [22], problem diagnosis [27], and reconfigurability [7]. While most end-host operating systems have interfaces for configuring certain limited aspects of network security and resource policy, these configurations are typically set independently by each user and therefore provide little assurance or consistent behavior when composed across multiple users on a network.

Instead, most network administrators turn to middleboxes - a central point of control at the edge of the network where functionality can be added and enforced on all users. Unfortunately, middleboxes are neither a complete nor a cost-efficient solution. Middleboxes are usually specialized appliances designed for a specific purpose, such as a firewall, packet shaper, or intrusion detection system, each with their own management interface and interoperability issues. Middleboxes are typically deployed at the edge of the (local area) network, providing no help to network administrators attempting to control behavior inside the network. Although middlebox functionality could conceivably be integrated with every network switch, doing so is not feasible at line-rate at reasonable cost with today's LAN switch hardware.

We propose a more direct approach, to manage network resources via software running in trusted execution environments on participating endpoints. Although the software is physically running on endpoints, it is logically controlled centrally by the network administrator. We somewhat whimsically call our approach ETTM, or End to the Middle. Of course, there is still a middle, to validate the trusted computing stack running on each participating node, and to redirect traffic originating from non-participating nodes such as smart phones and printers to a trusted intermediary on the network. By moving packet processing to trusted endpoints, we can enable a much wider variety of network management functionality than is possible with today's network-based solutions.

Our approach leverages four separate hardware and software trends. First, network switches increasingly have the ability to re-route or filter traffic under administrator control [7, 30]. This functionality was originally added for distributed access control, e.g., to prevent visitors from connecting to the local file server. We use these new-generation switches as a lever to a more general, fine-grained network control model, e.g., allowing us to efficiently interpose trusted network management software on every packet. Second, we observe that many end-host computers today are equipped with trusted computing hardware, to validate that the endpoint is booted with an uncorrupted software stack. This allows us to use software running on endpoints, and not just network hardware in the middle of the network, as part of our enforcement mechanism for network management. Third, we leverage virtual machines. Our

network management software runs in a trusted virtual machine which is logically interposed on each network packet by a hypervisor. Despite this, to the user each computer looks like a normal, completely configurable local PC running a standard operating system. Users can have complete administrative control over this OS without compromising the interposition engine. Finally, the rise of multicore architectures means that it is possible to interpose trusted packet processing on every incoming/outgoing packet without a significant performance degradation to the rest of the activity on a computer.

In essence, we advocate converting today's closed appliance model of network management to an open software model with a standard API. None of the functionality we need to implement on top of this API is particularly complex. As a motivating example, consider a network administrator needing to set up a computer lab at a university in a developing country with an underprovisioned, high latency link to the Internet. It is well understood that standard TCP performance will be dreadful unless steps are taken to manipulate TCP windows to limit the rate of incoming traffic to the bandwidth of the access link, to cache repeated content locally, and to prioritize interactive traffic over large background transfers. As another example, consider an enterprise seeking to detect and combat worm traffic inside their network. Current Deep Packet Inspection (DPI) techniques can detect worms given appropriate visibility, but are expensive to deploy pervasively and at scale. We show that it is possible to solve these issues in software, efficiently, scalably, and with high fault tolerance, avoiding the need for expensive and proprietary hardware solutions.

The rest of the paper discusses these issues in more detail. We describe our design in § 2, sketch the network services which we have built in § 3, summarize related work in § 4 and conclude in § 5.

2 Design & Prototype

ETTM is a scalable and fault-tolerant system designed to provide a reliable, trustworthy and standardized software platform on which to build network management services without the need for specialized hardware. However, this approach begs several questions concerning security, reliability and extensibility.

- How can network management tasks be entrusted to commodity end hosts which are notorious for being insecure? In our model, network management tasks can be relocated to any trusted execution environment on the network. This requires the network management software be verified and isolated from the host OS to be protected from compromise.
- If the management tasks are decentralized, how can these distributed points of control provide consistent decisions which survive failures and disconnections? The system should not break simply because a user, or a whole team of users, turn off their computers. In particular, management services must be available in face of node failures and maintain consistent state regarding the resources they manage.
- How can we architect an extensible system that enables the deployment of new network management services which can interpose on relevant packets? Network administrators need a single interface to install, configure and compose new network management services. Further, the implementation of the interface should not impose undue overheads on network traffic.

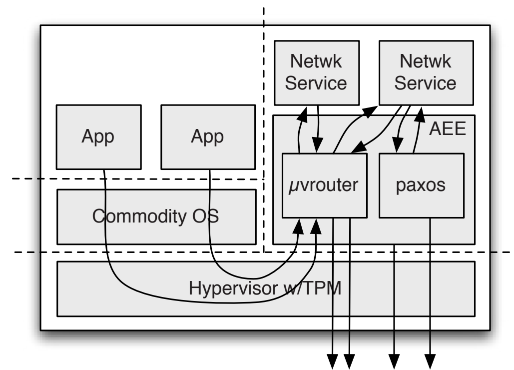


Figure 1: The architecture of an ETTM end-host. Network management services run in a trusted virtual machine (AEE). Application flows are routed to appropriate network management services using a micro virtual router (μvrouter).

While many of the techniques we employ to surmount these challenges are well-known, their combination into a unified platform able to support a diverse set of network services is novel. The particular mechanisms we employ are summarized in Table 1, and the architecture of a given end-host participating in management can be seen in Figure 1.

The function of these mechanisms is perhaps best illustrated by example, so let us consider a distributed Network Address Translation (NAT) service for sharing a single IP address among a set of hosts. The NAT service in ETTM maps globally visible port numbers to private IP addresses and vice versa. First, the translation table itself needs to be consistent and survive faults, so it is maintained and modified consistently by the *consensus* subsystem based on the Paxos distributed coordination algorithm. Second, the translator must be able to interpose on all traffic that is either entering or leaving the NATed network. The micro virtual router (μvrouter)'s *filters* allow for this interposition on packets sourced by a ETTM end-host, while the *physical switches* are set up to

Mechanism	Description	Tech Trends	Goals	Section
Trusted Authorization	Extension to the 802.1X network access control protocol to authorize trusted stacks	TPM	Trust	2.1
Attested Execution Environment	Trusted space to run filters and control plane processes on untrusted end-hosts	Virtualization, Multicore	Scalability	2.2
Physical Switches	In-network enforcers of access control and routing/switching policy decisions	Open interfaces	Standardization	2.3
Filters	End-host enforcers of network policy running inside the Attested Execution Environment	Multicore	Extensibility	2.4
Consensus	Agreement on management decisions and shared state	Fault tolerance techniques	Reliability, Extensibility	2.5

Table 1: Summary of mechanisms in ETTM.

deliver incoming packets to the appropriate host.[1] Lastly, because potentially untrusted hosts will be involved in the processing of each packet, the service is run only in an isolated *attested execution environment* on hosts that have been verified using our *trusted authorization* protocol based on commodity trusted hardware.

2.1 Trusted Authorization

Traditionally, end-hosts running commodity operating systems have been considered too insecure to be entrusted with the management of network resources. However, the recent proliferation of trusted computing hardware has opened the possibility of restructuring the placement of trust. In particular, using the trusted platform module (TPM) [39] shipping with many current computers, it is possible to verify that a remote computer booted a particular software stack. In ETTM, we use this feature to build an extension to the widely-used 802.1X network access control protocol to make authorization decisions based on the booted software stack of end-hosts rather than using traditional key- or password-based techniques. We note that the guarantees provided by trusted computing hardware generally assume that an attacker will not physically tamper with the host, and we make this assumption as well.

The remainder of this section describes the particular capabilities of current trusted hardware and how they enable the remote verification of a given software stack.

2.1.1 Trusted Platform Module

The TPM is a hardware chip commonly found on motherboards today consisting of a cryptographic processor, some persistent memory, and some volatile memory. The TPM has a wide variety of capabilities including the secure storage of integrity measurements, RSA key creation and storage, RSA encryption and decryption of data, pseudo-random number generation and attestation to portions of the TPM state. Much of this functionality is orthogonal to the purposes of this paper. Instead, we focus on the features required to remotely verify that a machine has booted a given software stack.

One of the keys stored in the TPM's persistent memory is the endorsement key (EK). The EK serves as an identity for the particular TPM and is immutable. Ideally, the EK also comes with a certificate from the manufacturer stating that the EK belongs to a valid hardware TPM. However many TPMs do not ship with an EK from the manufacturer. Instead, the EK is set as part of initializing the TPM for its first use.

The volatile memory inside the TPM is reset on every boot. It is used to store measurement data as well as any currently loaded keys. Integrity measurements of the various parts of the software stack are stored in registers called Platform Configuration Registers (PCRs). All PCR values start as 0 at boot and can only be changed by an extend operation, i.e., it is not possible to replace the value stored in the PCR with an arbitrary new value. Instead, the extend operation takes the old value of the PCR register, concatenates it with a new value, computes their hash using Secure Hash Algorithm 1 (SHA-1), and replaces the current value in the PCR with the output of the hash operation.

2.1.2 Trusted Boot

The intent is that as the system boots, each software component will be hashed and its hash will be used to extend at least one of the PCRs. Thus, after booting, the PCRs will provide a tamper evident summary of what happened during the boot. For instance, the post-boot PCR values can be compared against ones corresponding to a known-good boot to establish if a certain software stack has been loaded or not.

To properly measure all of the relevant components in the software stack requires that each layer be instrumented to measure the integrity of the next layer, and then store that measurement in a PCR before passing execution on. Storing measurements of different components into different PCRs allows individual modules to

[1] This is possible with legacy ethernet switches using a form of detour routing or more efficiently with programmable switches [30].

be replaced independently.

As each measurement's validity depends on the correctness of the measuring component, the PCRs form a chain of trust that must be rooted somewhere. This root is the immutable boot block code in the BIOS and is referred to as the Core Root of Trust for Measurement (CRTM). The CRTM measures itself as well as the rest of BIOS and appends the value into a PCR before passing control to any software or firmware. This means that any changeable code will not acquire a blank PCR state and cannot forge being the "bottom" of the stack.

It should be noted that the values in the PCRs are only representative of the state of the machine at boot time. If malicious software is loaded or changes are made to the system thereafter, the changes will not be reflected in the PCRs until the machine is rebooted. Thus, it is important that only minimal software layers are attested. In our case, we attest the BIOS, boot loader, virtual machine monitor, and execution environment for network services. We do not need to attest the guest OS running on the device, as it is never given access to the raw packets traversing the device.

2.1.3 Attestation

Once a machine is booted with PCR values in the TPM, we need a verifiable way to extract them from the TPM so that a remote third party can verify that they match a known-good software stack and that they came from a real TPM. In theory this should be as simple as signing the current PCR values with the private half of the EK, but signing data with the EK directly is disallowed.[2] Instead, Attestation Identity Keys (AIKs) are created to sign data and create attestations. The AIKs can be associated with the TPM's EK either via a Privacy CA or via Direct Anonymous Attestation [39] in order to prove that the AIKs belong to a real TPM. As a detail, because many TPMs do not ship with EKs from their manufacturers, these computers must generate an AIK at installation and store the public half in a persistent database.

To facilitate attestation, TPMs provide a quote operation which takes a nonce and signs a digest of the current PCRs and that nonce with a given AIK. Thus, a verifier can challenge a TPM-equipped computer with a random, fresh nonce and validate that the response comes from a known-good AIK, contains the fresh nonce, and represents a known-good software stack.

2.1.4 ETTM Boot

When a machine attempts to connect to an ETTM network, the switch forwards the packets to a verification server which can be either an already-booted end-host running ETTM, a persistent server on the LAN or even a cloud service.[3] On recognizing the connection of a new host, the switch establishes a tunnel to the verification server and maintains this tunnel until the verification server can reach a verdict about authorization.

[2]This is to avoid creating an immutable identity which is revealed in every interaction involving the TPM.

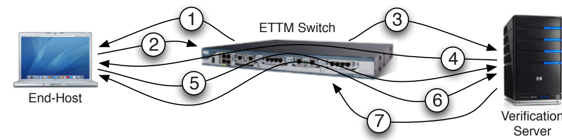


Figure 2: The steps required for an ETTM boot and trusted authorization.

If the host is verified as running a complete, trusted software stack then it is simply granted access to the network. If the host is running either an incomplete or old software stack, the ETTM software on the end-host attempts to download a fresh copy and retries. Traffic from non-conformant hosts are tunneled to a participating host; our design assumes this is a rare case.

Our trusted authorization protocol creates this exchange via an extension to the 802.1X and EAP protocols. We have extended the `wpa_supplicant` [28] 802.1X client and the FreeRADIUS [16] 802.1X server to support this extension and provide authorization to clients based purely on their attested software stacks.

This process is shown in Figure 2. First, the end-host connects to an ETTM switch, receives an EAP Request Identity packet (1), and responds with an EAP Response/Identity frame containing the desired AIK to use (2). The switch encapsulates this response inside an 802.1x packet which is forwarded to the verification server running our modified version of FreeRADIUS (3). The FreeRADIUS server responds with a second EAP Request Trusted Software Stack frame containing a nonce again encapsulated inside an 802.1X packet (4), and the end-host responds with an EAP Response Trusted Software Stack frame containing the signed PCR values proving the booted software stack (5). This concludes the verification stage.

The verification server can then either render a verdict as to whether access is granted (7) or require the end-host to go through a provisioning stage (6) where extra code and/or configuration can be loaded onto the machine and the authorization retried.

2.1.5 Performance of ETTM Boot

Table 2 presents microbenchmarks for various TPM operations (including those which will be described later in this section) on our Dell Latitude e5400 with a Broadcom TPM complying to version 1.2 of the TPM spec, an Intel 2 GHz Core 2 Duo processor and 2 GB of RAM.

[3]We assume the existence of some persistently reachable computer to bootstrap new nodes and store TPM configuration state. Under normal operation, this is a currently active verified ETTM node.

Operation	Time (s)	Std. Dev. (s)
PCR Extend	0.0253	0.001
Create AIK	34.3	8.22
Load AIK	1.75	0.002
Sign PCR	0.998	0.001

Table 2: The time (in seconds) it takes for a variety of TPM operations to complete.

Operation	Wall Clock Time (s)
client start	0
receive first server message	+0.049
receive challenge nonce	+0.021
send signed PCRs	+0.998
receive server decision	+0.018
Total	1.09

Table 3: The time (in seconds) it takes for an 802.1X EAP-TSS authorization with breakdown by operation.

The time to create the AIK is needed only once at system initialization. The total time added to the normal boot sequence for an ETTM enabled host is negligible as most actions can be trivially overlapped with other boot tasks. Assuming the challenge nonce is received, the signing time can be overlapped with the booting of the guest OS as no attestation is required to its state.

Table 3 shows a breakdown of how long each step takes in our implementation of trusted authorization assuming an up-to-date trusted software stack is already installed on the end-host and the relevant AIK has already been loaded. The total time to verify the boot status is just over 1 second. This is dominated by the time that it takes to sign the PCR values after having received the challenge nonce.

2.2 Attested Execution Environment

In ETTM, we require that each participating host has a corresponding trusted virtual machine which is responsible for managing that host's traffic. We call this virtual machine an Attested Execution Environment (AEE) because it has been attested by Trusted Authorization. In the common case, this virtual machine will run alongside the commodity OS on the host, but in some cases a host's corresponding AEE may run elsewhere with the physical switching infrastructure providing an constrained tunnel between the host and its remote VM.

The AEE is the vessel in which network management activities take place on end-hosts. It provides three key features: a mechanism to intercept all incoming and outgoing traffic, a secure and isolated execution environment for network management tasks and a common platform for network management applications.

To interpose the AEE on all network traffic, the hypervisor (our implementation makes use of Xen [3]) is configured to forward all incoming and outgoing network traffic through the AEE. This configuration is verified as part of trusted authorization. Once the AEE has been interposed on all traffic, it can apply the ETTM filters (described in § 2.4) giving each network service the required points of visibility and control of the data path.

Further, the hypervisor is configured to isolate the AEE from any other virtual machines it hosts. Thus, the AEE will be able to faithfully execute the prescribed filters regardless of the configuration of the commodity operating system. [4] The AEE can also execute network management tasks which are not directly related to the host's traffic. For example, it could redirect traffic to a mobile host, verify a new host's software stack or reconfigure physical switches. It is even possible for a host to run multiple AEEs simultaneously with some being run on behalf of other nodes in the system. A desktop with excess processing power can stand-in to filter the traffic from a mobile phone.

Lastly, the AEE provides a common platform to build network management services. Because this platform is run as a VM, it can remain constant across all end-hosts providing a standardized software API. Our current AEE implementation is a stripped-down Linux virtual machine, however, we have augmented it with APIs to manage filters (described in § 2.4) as well as to manage reliable, consistent, distributed state (described in § 2.5).

While in most cases, the added computational resources required to run an AEE do not pose a problem, ETTM allows for AEEs (or some parts of an AEE) to be offloaded to another computer. In our prototype, this is handled by applications themselves. In the future, we hope to add dynamic offloading based on machine load.

2.3 Physical Switches

Physical switches are the lowest-level building block in ETTM. Their primary purpose is to provide control and visibility into the link layer of the network. This includes access control, flexible control of packet forwarding, and link layer topology monitoring.

- **Authorization/Access Control:** As described earlier, switches redirect and tunnel traffic from as of yet unauthorized hosts until an authorization decision has been made.
- **Flexible Packet Forwarding:** The ability to install custom forwarding rules in the network enables significantly more efficient implementations of some network management services (e.g., NAT), but is not required. Flexible forwarding also enables more efficient routing by not constraining traffic within the

[4] We make use of a VM other than the root VM (e.g., Dom0 in Xen) for the AEE to both maintain independence from any particular hypervisor and to protect any such root VM from misbehaving applications in the AEE.

traditional ethernet spanning tree protocol.

- **Topology Monitoring:** In order to properly manage available network resources, end-hosts must be able to discover what network resources exist. This includes the set of physical switches and links along with the links' latencies and capacities.

At a minimum, ETTM only requires the first of these capabilities and since we implement access control via an extension to 802.1X and EAP, most current ethernet switches (even many inexpensive home routers [31, 10]) can serve as ETTM switches. There are advantages to more full-featured switches, however. For instance, a physical switch that supports the 802.1AE MACSec specification can provide a secure mechanism to differentiate between the different hosts attached to the same physical port and authorize them independently, while denying access to other unauthorized hosts attached to the port.

Additionally, ETTM can better manage network resources when used in conjunction with an OpenFlow switch [30]. OpenFlow provides a wealth of network status information and supports packet header rewriting and flexible, rule-based packet forwarding. We currently leave interacting with programmable switches to applications. Many applications function correctly using simple Ethernet spanning tree routing and do not require control over packet-forwarding. Those that do, like the NAT, must either implement packet redirection in the application logic by having AEEs forward packets to the appropriate host or manage configuring the programmable switches themselves. We are in the process of creating a standard interface to packet forwarding in ETTM.

2.4 Micro Virtual Router

On each end-host, we construct a lightweight virtual router, called the micro virtual router (μvrouter), which mediates access to incoming and outgoing packets by the various services. Services use the μvrouter to inspect and modify packets as well as insert new packets or drop packets. The core idea of filters in ETTM is that they are the mechanism to interpose on a per-packet basis and their behavior can be controlled by consensus operations which occur at a slower time scale: one operation per flow or one operation per flow, per RTT.

The μvrouter consists of an ordered list (by priority) of filters which are applied to packets as they depart and arrive at the host. The current Filter API is described in Table 4. The filters which we have implemented so far (described in § 3) correspond to tasks that would currently be carried out by a special-purpose middlebox like a NAT, web cache, or traffic shaper.

The μvrouter is approximately 2250 lines of C++ code running on Linux using `libipq` and `iptables` to capture traffic. This has simplified development by allowing

`matchOnHeader()` returns `true` if the filter can match purely on IP, TCP and UDP headers (i.e., without considering the payload) and `false` if the filter must match on full packets
`getPriority()` returns the priority of the filter, this is used to establish the order in which filters are applied
`getName()` simply returns a human readable name of the filter
`matchHeader(iphdr, tcphdr, udphdr)` returns `true` if the filter is interested in a packet with these headers; undefined filters are set to `NULL` and behavior is undefined if `matchOnHeader()` returns `false`
`match(packet)` returns `true` if the filter is interested in the packet; behavior is undefined if `matchOnHeader()` returns `true`
`filter(packet)` actually processes a packet; returns one of `ERROR`, `CONTINUE`, `SEND`, `DROP` or `QUEUED` and possibly modifies the packet
`upkeep()` this function is called 'frequently' and enables the filter to perform any maintenance that is required
`getReadyPackets()` this returns a list of packets that the filter would like to either dequeue or introduce; this is called 'frequently'

Table 4: The filter API.

the μvrouter to run as a user-space application. However, the user-space implementation has a downside in that it imposes performance overheads that limit the sustained throughput for large flows. To address the performance concerns, we split the functionality of the μvrouter into two components—a user-space module supporting the full filter API specified in Table 4 and a kernel-level module that supports a more restricted API used only for header rewriting and rate-limiting. In applications such as the NAT, the user-space filter is invoked only for the first packet in order to assign a globally unique port number to the flow, while the kernel module is used for filling in this port number in subsequent packets.

The μvrouter enables an administrator to specify a stack of filters that carry out the data-plane management tasks for the network. That is, it handles traffic that is destined for or emanates from an end-host on the network. Traffic destined to or emanating from AEEs or physical switches constitutes the control plane of ETTM and is not handled by the filters.

2.5 Consensus

If network management is going to be distributed among a large number of potentially unreliable commodity computers, there must be a layer to provide consistency and reliability despite failures. For example, a desktop unexpectedly being unplugged should not cause any state to

be lost for the remaining functioning computers. Fortunately, there is a vast literature on how to build reliable systems out of inexpensive, unreliable parts. In our case we build reliability using the Paxos algorithm for distributed consensus [25].

We expose a common API which provides a simple way for ETTM network services to manage their consistent state including the ability to define custom rules for what state should be semantically allowed and ways to choose between liveness and safety in the event that it is required. We expose our consensus implementation via a table abstraction in which each row corresponds to a single service's state and each cell in a given row corresponds to an agreed upon action on the state managed by the service. Thus, each service has its own independently ordered list of agreed upon values, with each row entirely independent of other rows from the point of view of the Paxos implementation.

In building the API and its supporting implementation we strove to overcome several key challenges:

Application Independent Agreement: The actual agreement process should be entirely independent of the particular application. As a consequence, the abstraction presented is agreement on an ordered list of blobs of bytes for each application or service, with the following operations allowed on this ordered list.

- `put(name, value)`: Attempts to place `value` as a cell in the row named `name`. This will not return immediately specifying success or failure, but if the value is accepted, a later `get` call or subscription will return `value`.
- `get(name, seqNum)`: Attempts to retrieve cell number `seqNum` from the row named `name`. Returns an error if `seqNum` is invalid and the relevant value otherwise.

For example, our NAT implementation creates a row in the table called "NAT". When an outgoing connection is made an entry is added with the mapping from the private IP address and port to the public IP address and a globally visible port along with an expiration time. Nodes with long-running connections can refresh by appending a new entry. Thus, each node participating in the NAT can determine the shared state by iteratively processing cells from any of the replicas.

Publish-Subscribe Functionality: A network service can subscribe to the set of agreed upon values for a row via the `subscribe` API call. The service running on an ETTM node receives a callback (using `notify`) when new values are added to a given row through the `put` API calls. This is useful not just for letting services manage their own state, but also for subscribing to special rows that contain information about the network in general. For instance, there is one row which describes topology information and another row which logs authorization decisions. The consensus system invokes

- `subscribe(name, seqNum)`: Asks that the values of all cells in the row `name` starting with the cell numbered `seqNum` be sent to the caller. This includes all cells agreed on in the future.
- `unsubscribe(name)`: Cancels any existing subscription to the row `name`.
- `notify(name, value, seqNum)`: This is the callback from a `subscription` call and lets the client know that cell number `seqNum` of row `name` has the value `value`.

Balance Reliability and Performance: Invariably adding more nodes and thus increasing expected reliability causes performance to degrade as more responses are required. Thus, we allow for a subset of the participating ETTM nodes to form the Paxos group rather than the whole set. ETTM nodes use the following API calls to join and depart from consensus groups and to identify the set of cells that have been agreed upon by the consensus group.

- `join(name)` Asks the local consensus agent to participate in the row `name`.
- `leave(name)` Asks the local consensus agent to stop participating in row `name`. A graceful ETTM machine shutdown involves informing each row that the node is leaving beforehand.
- `highestSequenceNumber(name)` Returns the current highest valid cell number in the row named `name`.

Allow Application Semantics: While we wish to be application agnostic in the details of agreement, we also would like services to be able to enforce some semantics about what constitute valid and invalid sequences of values. Coming back to the NAT example, the semantic check can ensure that a newly proposed IP-port mapping does not conflict with any previously established ones and can even deal with the leased nature of our IP-port mappings making the decision once (typically at the leader of the Paxos group) as to whether the old lease has expired or not. We accomplish this by having network services optionally provide a function to check the validity of each value before it is proposed.

- `setSemanticCheckPolicy(name, policyhandler)`: Sets the semantic check policy for row `name`. `policyhandler` is an application-specific call-back function that is used to check the validity of the proposed values.
- `check(policyhandler, name, value, seqNum)`: Asks the consensus client if `value` is

a semantically valid value to be put in cell number `seqNum` of row `name`. Returns `true` if the value is semantically valid, `false` if it is not and with an error if the checker has not been informed of all cells preceding cell number `seqNum`.

Finally, each row maintained by the consensus system can have a different set of policies about whether to check for semantic validity, whether to favor safety or liveness (as described below), and even which nodes are serving as the set of replicas.

2.5.1 Catastrophic Failures

Paxos can make progress only when a majority of the nodes are online. If membership changes gradually, the Paxos group can elect to modify its membership. The two critical parameters that determine the robustness of the quorum are the churn rate and the time it takes to detect failure and change the group's membership. The consensus group can continue to operate if fewer than half of the nodes fail before their failure is detected. In such cases, since a majority of the machines in the consensus group are still operating, we have that set vote on any changes necessary to cope with the churn [26].

But if a large number of nodes leave simultaneously (e.g., because of a power outage), we allow services to opt to make progress despite inconsistent state. Each service can pick they want to handle this case for its row, deciding to either favor liveness or safety via the `setForkPolicy` call. If the row favors safety, then the row is effectively frozen until a time when a majority of the nodes recover and can continue to make progress. However, we allow for a row to favor liveness, in which case the surviving nodes make note of the fact that they are potentially breaking safety and fork the row.

Forking effectively creates a new row in which the first value is an annotation specifying the row from which it was forked off, the last agreed upon sequence number before the fork and the new set of nodes which are believed to be up. This enables a minority of the nodes to continue to make progress. Later on, when a majority of the nodes in the original row return to being up, it is up to the service to merge the relevant changes (and deal with any potential conflicts) from the forked row back into the main row via the normal `put` operation and eventually garbage collect the forked row via a `delete` operation. The details of this API are described in Table 5.

While, in theory, building services that can handle potentially inconsistent state is hard, we have found that, in practice, many services admit reasonable solutions. For instance, a NAT which experiences a catastrophic failure can continue to operate and when merging conflicts it may have to terminate connections if they share the same external IP and port, though most of the time there will be no such conflicts.

API
`setForkPolicy(name, policy)` Sets the forking policy for the row `name` in the case of catastrophic failures. The valid values of `policy` are 'safe' and 'live'.
`delete(name)` Cleans up the state associated with row `name`. Fails if called on a row which is not a fork of an already existing row.
`forkNotify(name, forkName)` Informs the consensus client that because the client asked to favor liveness over safety, the row `name` has been forked and that a new copy has been started as row `forkName` where potentially unsafe progress can be made, but may need to be later merged.

Table 5: API for dealing with catastrophic failures.

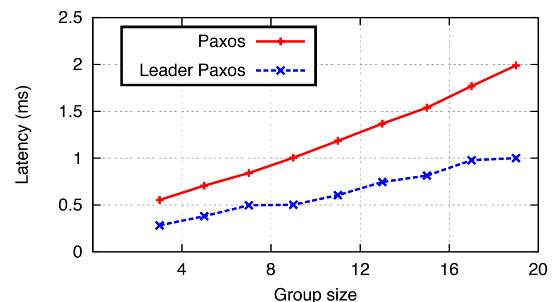


Figure 3: The average time for a Paxos round to complete with and without a leader as we vary the size of the Paxos group.

2.5.2 Implementation

Our current implementation of consensus is approximately 2100 lines of C++ code implementing a straightforward and largely unoptimized adaptation of the Paxos distributed agreement protocol. In Paxos, proposals are sent to all participating nodes and accepted if a majority of the nodes agree on the proposal. In our implementation, one leader is elected per row and all requests for that row are forwarded to the leader. If progress stalls, the leader is assumed to have failed and a new one is elected without concern for contention. If progress on electing a leader stalls, then the row can be unsafely forked depending on the requested forking policy. As nodes fail, the Paxos group reconfigures itself to remove the failed node from the node set and replace it with a different ETTM end-host.

Figure 3 shows the average time for a round of our Paxos implementation to complete when running with varying numbers of `pc3000` nodes (with 3GHz, 64-bit Xeon processors) on Emulab [15]. The results show that a Paxos round can be completed within 2 ms when there is no leader and within 1 ms with a leader. While the computation necessarily grows linearly with the number of nodes, this effect is mitigated by running Paxos on a subset of the active ETTM nodes. For example, as we

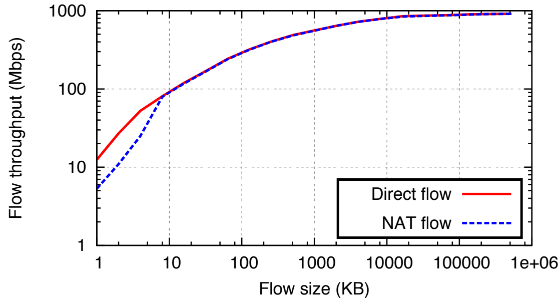


Figure 4: Bandwidth throughput of flows traversing ETTM NAT as we vary the flow size.

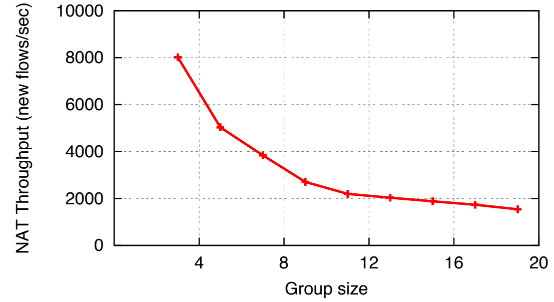


Figure 5: Throughput performance of ETTM NAT as we vary the Paxos group size.

will show in our evaluation of the NAT, a Paxos group of only 10 nodes—with new machines brought in only to replace any departing nodes in the subset—provides sufficient throughput and availability for the management of a large number of network flows.

3 Network Management Services

We next describe the design, implementation, and evaluation of several example services we have built using ETTM. These services are intended to be proof of concept examples of the power of making network administration a software engineering, rather than a hardware configuration, problem. In each case the functionality we describe can also be implemented using middleboxes. However, a centralized hardware solution increases costs and limits reliability, scalability, and flexibility. Proposals exist to implement several of these services as peer-to-peer applications on end-hosts [23, 38], but this raises questions of enforcement and privacy. Instead, ETTM provides the best of both worlds: safe enforcement of network management without the limitations of hardware solutions.

3.1 NATs

Network Address Translators (NATs) share a single externally-visible IP address among a number of different hosts by maintaining a mapping between externally visible TCP or UDP ports and the private, internally-visible IP addresses belonging to the hosts. Mappings are generated on-demand for each new outgoing connection, stored and transparently applied at the NAT device itself. Traffic entering the network which does not belonging to an already-established mapping is dropped. As a result, passive listeners such as servers and peer-to-peer systems can have connectivity problems when located behind NATs. Mappings are usually not replicated, so a rebooted NAT will break all connections.

In contrast, Our ETTM NAT is distributed and fault-tolerant. We store the mappings using the consensus API allowing any participating AEE to access the complete list of mappings. When the NAT filter running in a host's AEE detects a new outgoing flow, it temporarily hold the flow and requests a mapping to an available, externally-visible port. This request is satisfied only if the port is actually available. Once this request completes, the NAT filter begins rewriting the packet headers for the flow and allows packets to flow normally.

Handling incoming traffic is slightly more complicated. If the physical switches on the network support flexible packet forwarding (as with OpenFlow hardware), they can be configured with soft state to forward traffic to the appropriate host where its NAT filter can rewrite the destination address.[5] If the soft state has not yet been installed or has been lost due to failure, default forwarding rules result in the packet being delivered to some host which can appropriately forward the packet and install rules in the physical switches as needed.

Our NAT also works if the physical switches do not support re-configurable routing. Instead, we assign the globally-visible IP address to a specific AEE and have that AEE forward traffic to appropriate hosts. While this might appear to be similar to proxying all external traffic through an end-host, such an approach would be neither fault tolerant nor privacy preserving. In contrast, in ETTM the AEE allows for packets to be silently redirected to the appropriate host without those packets being visible to the user of the forwarding host. Also, the failure of that AEE can be detected and another can be chosen with no lost state. When selecting an AEE, we use historical uptime data as well as information about current load to avoid using unreliable hosts and to avoid unnecessarily burdening loaded hosts. While it is possible that a determined snoop might physically tap their ethernet wire to see forwarded packets, deployments that wish to prevent this could enforce end-to-end encryption using a combination of SSL, IPsec and/or 802.1AE MACsec to encrypt all traffic entering or exiting the organization.

Our NAT can be configured to allow passive connec-

[5] We implement address translation in the AEE despite OpenFlow support because some of our OpenFlow hardware has worse performance when modifying packets. Further, keeping translation tables reliably in AEEs keeps no hard state in the network.

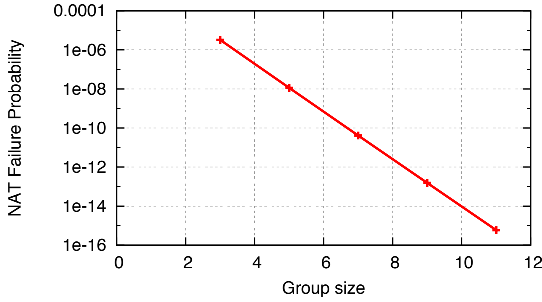
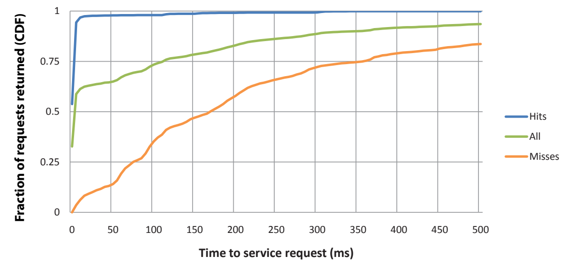
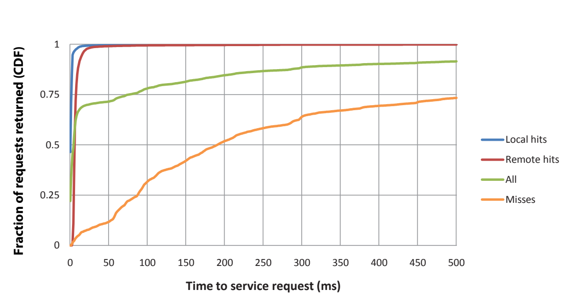


Figure 6: Availability of ETTM NAT as we vary the Paxos group size. Note the y-axis is in log scale.



(a) Latency by request type with a single centralized cache.



(b) Latency by request type with a distributed cache across 6 nodes.

Figure 7: The cumulative distribution of latencies by type of request with a centralized (Figure 7(a)) and distributed (Figure 7(b)) web caches.

tions to establish mappings. We have implemented a Linux kernel module that can be installed in the guest OS to explicitly notify the NAT filter whenever `bind()` or `listen()` is called, triggering a request for a valid mapping to an external IP address and port. This allows the ETTM system to direct incoming connections to the appropriate host without having the administrator set up customized port forwarding rules. We attempt to provide passive connections with the same external port as its internal one; if this is not possible, the kernel module can be queried for the external port number.

Note that the ETTM approach for implementing NATs reinstates the fate sharing principle. We trivially support multiple ingress points to the network because there is no hard state stored in the network. A connection only fails if either endpoint fails or there is no path between them, but not if the middlebox fails. Even if the consensus group fails entirely, existing flows will still continue as long as one member of the group remains; of course, new flows may be delayed in this case.

We evaluated the performance of our NAT module on a cluster of `pc3000` nodes on Emulab. Figure 4 depicts the flow throughputs with and without the NAT module for TCP flows of various sizes over a 1 Gbps LAN link. The NAT filter imposes some added cost in terms of the latency of the first packet (about 1-2 ms), which affects the throughput of short flows in the LAN. For all other flows, the throughput of the NAT filter matches that of the direct communications channel, and it achieves the maximum possible throughput of 1 Gbps for large flows.

Figure 5 plots the throughput of ETTM NAT by measuring the number of NAT translations that it can establish per second as we vary the size of the Paxos group operating on behalf of the NAT. While the throughput falls with the number of nodes, it is still able to sustain an admission rate of 2000 new flows per second even with large Paxos groups. Additional scalability would be possible if the external port space were partitioned among multiple Paxos groups.

We also model the NAT failure probability using end-host availability data collected for hosts within the Microsoft corporate network [12, 5]. The trace data has 81% of the end-hosts available at any time, and the median session length of these end-hosts was in excess of 16 hours. Figure 6 plots the probability of catastrophic failures assuming independent failures and a generous failure detection and group reconfiguration delay of 1 minute. As we can see from this analysis, a handful of end-systems would suffice for most enterprise settings.

3.2 Transparent Distributed Web Cache

It is common for large networks to employ a transparent web cache such as Akamai [1] or squid [38] to improve performance and reduce bandwidth costs. These caches exploit similarity in different users' browsing habits to reduce the total bandwidth consumption while also improving throughput and latency for requests served from the cache.

Even though a shared cache is often very effective, many small and medium sized networks do not use one because of the administrative overhead of setting it up and the potential performance bottleneck if the centralized cache is misconfigured. An alternative is to coordinate caches on each end-host [23], but this requires reconfiguration by each user and it raises privacy concerns since requests can be snooped by anyone with administrative privileges on any machine.

We implemented a distributed and privacy preserving

distributed cache. The cache runs as an ETTM network management service that is triggered by a μvrouter filter capturing all traffic headed to port 80. The service first checks the local AEE's web cache to see if the request can be served from the local host. If it cannot be served locally, the service computes a consistent hash of the request url and forwards it to a participating remote AEE based on the computed hash value. If the remote AEE does not have the content cached, it retrieves the content from the origin server, stores a copy in its local cache, and returns the fetched content to the requesting node. Note that the protocol traffic in ETTM is captured by the web cache filter and is not visible to any of the guest OSes. Also, communication between the caches can be optionally encrypted to prevent snooping. We adapted squid [38] to serve as the cache in each AEE and to provide the logic for interpreting http header directives, such as when to forward requests to the origin due to cache timeouts or outright disabling of caching.

We evaluated our end-host based web-cache implementation using a trace driven simulation. In order to generate trace data we aggregated the browser history of three of the authors and replayed the trace data on six nodes on Emulab [15]. In the centralized experiments, all clients but one have their cache disabled and were configured to send all requests to the one remaining active cache. In the distributed experiments each node runs its own cache. In the centralized case, the single cache is set to 600 MB, while in the distributed experiments the cache size for each of the six nodes is set to 100 MB.

Cache hit rates are similar in both cases. For brevity we omit detailed analysis of hit rates and instead focus on latency. The cumulative distribution of latencies for the centralized and distributed caches is shown in Figure 7. The latency for objects found in the other node's caches is at most a few milliseconds more than local cache hits, indicating that the distributed nature of our implementation imposes little or no performance penalty.

3.3 Deep Packet Inspection

The ability to filter traffic based on the full packet contents and often the contents of multiple packets—commonly called deep packet inspection (DPI)—has quickly become a standard tool alongside traditional firewalls and intrusion detection systems for detecting security breaches. However, the computation required for deep packet inspection is still limits its deployment.

The ETTM approach opens the door to 'outsourcing' the DPI computation to end-hosts where there is almost certainly more aggregate compute power than inside a dedicated DPI middlebox. Traditionally, the idea of running this DPI code at end-hosts would flounder because they could not be trusted to execute the code faithfully—a virus infecting one host could undermine network security. While no security is invulnerable, we offer a narrow attack surface similar to middleboxes, and also use attestation to be able to make claims about booted software and detect malicious changes on reboots.

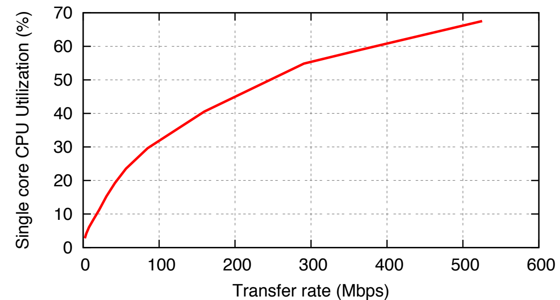


Figure 8: CPU load of ETTM DPI module as we vary the transfer rate of our trace.

Our implementation of DPI is based on the Snort [37] engine and renders decisions either by delaying or dropping traffic or by tagging flows with metadata. The DPI filter is run within the end-host AEE and inspects the flows being sourced from or received by the end-host. In addition, the DPI modules running on end-hosts periodically exchange CPU load information with each other. In situations where the end-host CPU is overloaded, as in highly-loaded web servers, the flows are redirected to some other lightly loaded end-host running the ETTM stack in order to perform the DPI tasks.

The two commonly used applications of DPI are to detect possible attacks and to discover obfuscated peer-to-peer traffic. In the case of detecting attacks, the filter releases traffic after it has been scanned for attack signatures and found to be clean. If a flow is flagged as an attack, no further traffic is allowed, and the source is labeled as being believed to be compromised. In the case of obfuscated peer-to-peer traffic, normal traffic is passed through the DPI filter without delay, but when a flow is categorized as peer-to-peer the flow is labeled with metadata. The next section describes how we can use these labels to adjust priorities for peer-to-peer traffic.

Figure 8 shows benchmark results from a trace-based evaluation of our DPI filter. We ran the ETTM stack on a quad-core Intel Xeon machine with 4 GB of RAM where each core runs at 2 GHz. However, we only make use of one core as `snort-2.8` is single-threaded. The traces are from DEFCON 17 "capture the flag" dataset [13], which contain numerous intrusion attempts and serve as commonly used benchmarks for evaluating DPI performance. We vary the trace playback rate from 1x to 1024x and measured the CPU load imposed by our DPI filter at various traffic rates. Figure 8 shows the load on the ETTM CPU to analyze traffic to/from that CPU. This demonstrates that running DPI on a single core per host is feasible. Stated in other terms, the ETTM approach

of performing DPI computation on end-hosts scales with the number of ETTM machines; centralizing DPI computation on specialized hardware is more expensive and less scalable.

3.4 Bandwidth Allocation

The ability for ETTM to control network behavior on a packet granularity provides an opportunity for more efficient bandwidth management. In TCP, hosts increase their send rates until router buffers overflow and start dropping packets. As a result, it is well-known that the latency of short flows degrades whenever a congested link is shared with a bandwidth-intensive flow. Many large enterprises deploy hardware-based packet shapers at the edge of the network to throttle high bandwidth flows before they overwhelm the bottleneck link. In this subsection, we demonstrate a backwardly compatible software-based ETTM solution to this issue; we use this as an illustration of how ETTM can be used to improve quality-of-service in an enterprise setting.

We call our bandwidth allocation strategy *TCP with reservations* or TCP-R; the approach is similar to the explicit bandwidth signaling in ATM. In TCP-R, bandwidth allocations for the bottleneck access link are performed by a controller replicated using the consensus API. Endpoints managing TCP flows make bandwidth allocation requests to the controller, which responds with reservations for short periods of time. We next describe the logic executed end-hosts followed by the controller logic.

Endpoint: Whenever a new flow crossing the access link appears and every RTT after that, the bandwidth allocation filter on the local host issues a bandwidth reservation request to the controller. The request is for the maximum bandwidth the host needs, that can be allocated safely without causing queueing at the congested link. The controller responds with an allocation and a reservation for the subsequent round-trips.

Once the reservation has been agreed upon, the filter limits the flow to using that amount of bandwidth until it issues a subsequent reservation. The amount of the new reservation is based on the last RTT of behavior. Let $A_f(i-1)$ be the bandwidth allocated to flow f in period $i-1$, and let $U_f(i-1)$ be the bandwidth utilized by it during the period. Then it makes a reservation request $R_f(i)$ based on the following logic; this preserves TCP behavior for the portion of the path external to the LAN, while allowing for explicit allocation of the access link.

- If the flow used up its allocation, it asks the controller to provide it the maximum allowed by the TCP congestion window ($R_f(i) = cwnd/RTT$).
- If the flow did not use up its bandwidth allocation in the previous RTT, then it issues a new request for the lesser of the bandwidth it did use and the TCP congestion window, relinquishing its unused reservation ($R_f(i) = min(cwnd/RTT, U_f(i-1))$).

Controller: The controller allocates bandwidth among the reservation requests according to max-min fairness. It publishes the results by committing its allocation decision across the various controller instances using Paxos. Note that the actual reservation amount can be less than what was requested.

Periodically the controller processes the bandwidth requests and makes an allocation using the following scheme to achieve max-min fairness. It sorts the flows based on their requested bandwidth. Let $R_0 \leq R_2 \leq R_3 ... R_{k-2} \leq R_{k-1}$ be the set of sorted bandwidth requests, L be the link access bandwidth, and $A = 0$ be the allocated bandwidth at the beginning of each allocation round. The controller considers these requests in increasing order and the requested bandwidth or its fair share, whichever is lower. Concretely, for each flow j, it does the following: $A_j = min(R_j, \frac{L-A}{k-j})$ and sets $A = A + A_j$. Note that $\frac{L-A}{k-j}$ is the fair share of flow j after having allocated A bandwidth resources to the j flows considered before it.

In practice, because it takes some time to acquire a reservation, we leave some fraction of the link (10% in our implementation) unallocated and allow each flow to send a few packets (4 in our implementation) before receiving a reservation. Because the time to acquire a reservation (a millisecond or less) is smaller than most Internet round trip times, this avoids adversely affecting flows with increased latency.

TCP-R has many benefits over traditional TCP. It does not drive the bottleneck link to saturation, thereby avoiding losses and sub-optimal use of network resources. In particular, latency sensitive web traffic can obtain their share of the bandwidth resource even if there are simultaneous large background transfers.

This implementation of bandwidth allocation assumes that we are only managing the upload bandwidth of our access link. In the future, we will to extend our implementation to handle arbitrary bottlenecks as well as the allocation of incoming bandwidth.

Evaluation: Our evaluation illustrates the ability of the ETTM bandwidth allocator to provide a fair allocation to interactive web traffic. On Emulab, we set up an access link with a bottleneck bandwidth of 10 Mb/s and compared the latency of accessing `google.com` with and without background BitTorrent traffic that is generated by a different end-host in the network. Figure 9 depicts the webpage access latency at different points in time. When there is no competing traffic, the average access latency is 0.68 seconds. When there is competing traffic (during attempts 11 through 30), the average access

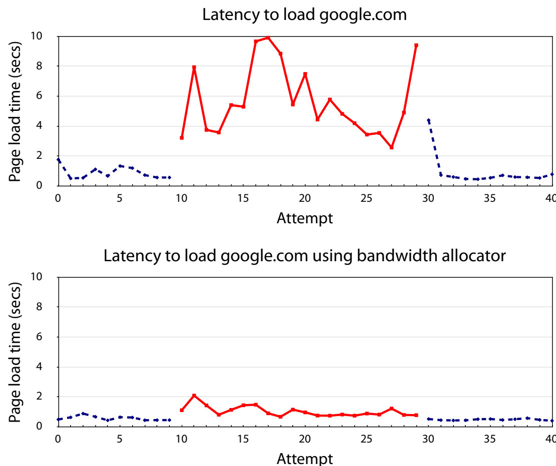


Figure 9: Webpage access latency in the presence of competing BitTorrent traffic with and without the bandwidth allocator. The solid lines depict the access latency when there is competing BitTorrent traffic.

latency is 5.67 seconds if we don't use the ETTM bandwidth allocator. With the ETTM bandwidth allocator, the interactive web traffic receives a fair share and incurs a latency of 1.04 seconds.

4 Related Work

Providing network administrators more control at lower cost is a longstanding goal of network research. Several recent projects have focused on providing administrators a logically centralized interface for configuring a distributed set of network routers and switches. Examples of this approach include 4D [34, 17, 42], NOX [19], Ethane [8, 7], Maestro [6] and CONMan [2]. Of course, the power of these systems is limited to the configurability of the hardware they control. While we agree with the need for logical centralization of network management functions, our hypothesis is that network administrators would prefer fine-grained, packet level control over their networks, something that is not possible at line-rate with today's current low cost network switches.

Other efforts have focused on building drop-in replacements for the the virtual ethernet switch inside existing hypervisors. Cisco's Nexus 1000V virtual switch [9, 40] provides a standard Cisco switch interface enabling switching policies to to the edge of VMs as well as hosts. Open vSwitch [33] accomplishes a similar feat, but provides an OpenFlow interface to the virtual switch and is compatible with Xen and a few other hypervisors. Still others are working to do hardware network I/O virtualization [32]. While all of these tools give network administrators additional points of control, they do not offer the flexibility required to implement the breadth of coordinated network polices administrators seek today. Instead, we are working to incorporate these standardized, simple points of control into ETTM to provide potentially higher performance some tasks and added control over the low-level network.

Other systems have tried to bring end-hosts into network management, though in limited ways. Microsoft's Active Directory includes Group Policy which allows for control over the actions which connected Windows hosts are allowed to carry out, but enforces them only assuming the host remains uncompromised. Network Exception Handlers [24] allow end-hosts to react to certain network events, but still leaves network hardware dominantly in control. Still other work [11] uses end-hosts to provide visibility into network traffic, but does not provide a point of control and assumes that the host remains uncompromised.

Other recent work has attempted to increase the flexibility of network switches to carry out administrative tasks. OpenFlow [30] adds the ability to configure routing and filtering decisions in LAN switches based on pattern matching on packet headers performed in hardware. A limitation of OpenFlow is throughput when packets need to be processed out of band, because there is typically only one underpowered control processor per LAN switch. In ETTM, we invoke out of band processing on the switch only for the initial TPM verification when the node connects, while still allowing the network administrator to add arbitrary processing on every packet.

Middleboxes have always been a contentious topic, but recent work has looked at how to embrace middleboxes and treat them as first-class citizens. In TRIAD [18] middleboxes are first-order constructs in providing a content-addressable network architecture. The Delegation-Oriented Architecture [41] allows hosts to explicitly invoke middleboxes, while NUTSS [20] proposes a novel connection establishment mechanism which includes negotiation of which middleboxes should be involved. Our work can be seen as enabling network administrators to place arbitrary packet-granularity middlebox functionality throughout the network, via validated software running on end-hosts.

Existing work has leveraged trusted computing hardware to avoid vulnerabilities in commodity software [35] as well as to ensure correct execution of specific tasks [29]. Our use of trusted computing hardware is complementary to these efforts.

5 Conclusion

Enterprise-level network management today is complex, expensive and unsatisfying: seemingly straightforward quality of service and security goals can be difficult to achieve even with an unlimited budget. In this paper, we have designed, implemented and evaluated a novel approach to provide network administrators more control at lower cost, and their users higher performance, more

reliability, and more flexibility. Network management tasks are implemented as software applications running in a distributed but secure fashion on every end-host, instead of on closed proprietary hardware at fixed points in the network. Our approach leverages the increasing availability of trusted computing hardware on end-hosts and reconfigurable routing tables in network switches, as well as the expansive computing capacity of modern multicore architectures. We show that our approach can support complex tasks such as fault tolerant network address translation, network-wide deep packet inspection for virus control, privacy preserving peer-to-peer web caching, and congested link bandwidth prioritization, all with reasonable performance despite the added overhead of fault tolerant distributed coordination.

Acknowledgements

We would like to thank our anonymous reviewers and our shepherd David Maltz for their valuable feedback. This work was supported in part by the National Science Foundation under grants NSF-0831540 and NSF-0963754.

References

[1] Akamai technologies. http://www.akamai.com/.

[2] Hitesh Ballani and Paul Francis. CONMan: A step towards network manageability. In *SIGCOMM*, 2007.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003.

[4] Blue Coat Systems. Blue Coat PacketShaper. http://www.bluecoat.com/products/packetshaper.

[5] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS*, 2000.

[6] Zheng Cai, Alan L. Cox, and T. S. Eugene Ng. Maestro: A new architecture for realizing and managing network controls. In *LISA Workshop on Network Configuration*, 2007.

[7] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[8] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A protection architecture for enterprise networks. In *USENIX Security*, 2006.

[9] Cisco Systems. Cisco Nexus 1000V Series Switches - Cisco Systems. http://www.cisco.com/en/US/products/ps9902/index.html.

[10] OpenFlow Consortium. OpenFlow >> OpenWrt. http://www.openflowswitch.org/wp/openwrt/.

[11] Evan Cooke, Richard Mortier, Austin Donnelly, Paul Barham, and Rebecca Isaacs. Reclaiming network-wide visibility using ubiquitous end system monitors. In *USENIX*, 2006.

[12] D. Narayanan and A. Donnelly and R. Mortier and A. Rowstron. Delay Aware Querying with Seaweed. In *VLDB*, 2006.

[13] Defcon 17 ctf packet traces. http://www.ddtek.biz/dc17.html.

[14] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT), 1994.

[15] Eric Eide, Leigh Stoller, and Jay Lepreau. An experimentation workbench for replayable networking research. In *NSDI*, 2007.

[16] FreeRADIUS: The world's most popular RADIUS Server. http://freeradius.org/.

[17] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. In *CCR*, 2005.

[18] Mark Gritter and David R Cheriton. An architecture for content routing support in the internet. In *USITS*, 2001.

[19] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. In *CCR*, 2008.

[20] Saikat Guha and Paul Francis. An end-middle-end approach to connection establishment. In *SIGCOMM*, 2007.

[21] Sotiris Ioannidis, Angelos D. Keromytis, Steve M. Bellovin, and Jonathan M. Smith. Implementing a distributed firewall. In *CCS*, 2000.

[22] RFC 3220: IP Mobility Support for IPv4, 2002.

[23] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, 2002.

[24] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. Network exception handlers: Host-network control in enterprise networks. In *SIGCOMM*, 2008.

[25] Leslie Lamport. The part-time parliament. *TOCS*, 16(2):133–169, 1998.

[26] Leslie Lamport. Paxos Made Simple. In *SIGACT*, 2001.

[27] Ratul Mahajan, Neil Spring, David Wetherall, and Thomas Anderson. User-level Internet Path Diagnosis. In *SOSP*, 2003.

[28] Jouni Malinen. Linux WPA Supplicant (IEEE 802.1X, WPA, WPA2, RSN, IEEE 802.11i). http://hostap.epitest.fi/wpa_supplicant/, January 2010.

[29] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys*, April 2008.

[30] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. http://www.openflowswitch.org/documents/openflow-wp-latest.pdf, March 2008.

[31] OpenWrt. http://openwrt.org/.

[32] PCI-SIG. PCI-SIG - I/O Virtualization. http://www.pcisig.com/specifications/iov/.

[33] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *HotNets*, 2009.

[34] Jennifer Rexford, Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Geoffrey Xie, Jibin Zhan, and Hui Zhang. Network-wide decision making: Toward a wafer-thin control plane. In *HotNets*, 2004.

[35] Seshadri, Arvind, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *SOSP*, 2007.

[36] S. Shenker, C. Partridge, and R. Guerin. RFC 2212: Specification of Guaranteed Quality of Service, 1997.

[37] Snort. http://www.snort.org.

[38] squid : Optimizing Web Delivery. http://www.squid-cache.org/.

[39] Trusted Computing Group. TPM Main Specification. http://www.trustedcomputinggroup.org/resources/tpm_main_specification, August 2007.

[40] VMware, Inc. Cisco Nexus 1000V Virtual Network Switch: Policy-Based Virtual Machine Networking. http://www.vmware.com/products/cisco-nexus-1000V/.

[41] Michael Walfish, Jeremy Stribling, Maxwell Krohn, Hari Balakrishnan, Robert Morris, and Scott Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.

[42] Hong Yan, David A. Maltz, T. S. Eugen Ng, Hemant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D network control plane. In *NSDI*, 2007.

Design, implementation and evaluation of congestion control for multipath TCP

Damon Wischik, Costin Raiciu, Adam Greenhalgh, Mark Handley
University College London

ABSTRACT

Multipath TCP, as proposed by the IETF working group `mptcp`, allows a single data stream to be split across multiple paths. This has obvious benefits for reliability, and it can also lead to more efficient use of networked resources. We describe the design of a multipath congestion control algorithm, we implement it in Linux, and we evaluate it for multihomed servers, data centers and mobile clients. We show that some 'obvious' solutions for multipath congestion control can be harmful, but that our algorithm improves throughput and fairness compared to single-path TCP. Our algorithm is a drop-in replacement for TCP, and we believe it is safe to deploy.

1. INTRODUCTION

Multipath TCP, as proposed by the IETF working group `mptcp` [7], allows a single data stream to be split across multiple paths. This has obvious benefits for reliability—the connection can persist when a path fails. It can also have benefits for load balancing at multihomed servers and data centers, and for mobility, as we show below.

Multipath TCP also raises questions, some obvious and some subtle, about how network capacity should be shared efficiently and fairly between competing flows. This paper describes the design and implementation of a multipath congestion control algorithm that works robustly across a wide range of scenarios and that can be used as a drop-in replacement for TCP.

In §2 we propose a mechanism for windowed congestion control for multipath TCP, and then spell out the questions that led us to it. This section is presented as a walk through the design space signposted by pertinent examples and analysed by calculations and thought experiments. It is not an exhaustive survey of the design space, and we do not claim that our algorithm is optimal—to even define optimality would require a more advanced theoretical underpinning than we have yet developed. Some of the issues (§2.1–§2.3) have previously been raised in the literature on multipath congestion control, but not all have been solved. The others (§2.4–§2.5) are novel.

In §3–§5 we evaluate our algorithm in three application scenarios: multihomed Internet servers, data centers, and mobile devices. We do this by means of simulations with a high-speed custom packet-level simulator, and with testbed experiments on a Linux implementation. We show that multipath TCP is beneficial, as long as congestion control is done right. Naive solutions can be worse than single-path TCP.

In §6 we discuss what we learnt from implementing the protocol in Linux. There are hard questions about how to avoid deadlock at the receiver buffer when packets can arrive out of order, and about the datastream sequence space versus the subflow sequence spaces. But careful consideration of corner cases forced us to our specific implementation. In §7 we discuss related work on protocol design.

In this paper we will restrict our attention to end-to-end mechanisms for sharing capacity, specifically to modifications to TCP's congestion control algorithm. We will assume that each TCP flow has access to one or more paths, and it can control how much traffic to send on each path, but it cannot specify the paths themselves. For example, our Linux implementation uses multihoming at one or both ends to provide path choice, but it relies on the standard Internet routing mechanisms to determine what those paths are. Our reasons for these restrictions are (i) the IETF working group is working under the same restrictions, (ii) they lead to a readily deployable protocol, i.e. no modifications to the core of the Internet, and (iii) theoretical results indicate that inefficient outcomes may arise when both the end-systems and the core participate in balancing traffic [1].

2. THE DESIGN PROBLEM FOR MULTIPATH RESOURCE ALLOCATION

The basic window-based congestion control algorithm employed in TCP consists of additive increase behaviour when no loss is detected, and multiplicative decrease when a loss event is observed. In short:

ALGORITHM: REGULAR TCP

- Each ACK, increase the congestion window w by $1/w$, resulting in an increase of one packet per RTT.[1]
- Each loss, decrease w by $w/2$.

Additionally, at the start of a connection, an exponential increase is used, as it is immediately after a retransmission timeout. Newer versions of TCP [24, 9] have

[1] For simplicity, we express windows in this paper in packets, but real implementations usually maintain them in bytes.

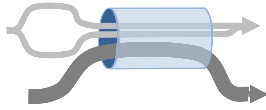


Figure 1: A scenario which shows the importance of weighting the aggressiveness of subflows.

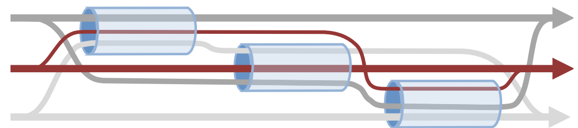


Figure 2: A scenario to illustrate the importance of choosing the less-congested path

faster behaviour when the network is underloaded; we believe our multipath enhancements can be straightforwardly applied to these versions, but it is a topic for further work.

The congestion control algorithm we propose is this:

ALGORITHM: MPTCP

A connection consists of set of subflows R, each of which may take a different route through the Internet. Each subflow $r \in R$ maintains its own congestion window w_r. An MPTCP sender stripes packets across these subflows as space in the subflow windows becomes available. The windows are adapted as follows:

- Each ACK on subflow r, for each subset $S \subseteq R$ that includes path r, compute
$$\frac{\max_{s \in S} w_s / \mathsf{RTT}_s^2}{\left(\sum_{s \in S} w_s / \mathsf{RTT}_s\right)^2}, \tag{1}$$
then find the minimum over all such S, and increase w_r by that much. (The complexity of finding the minimum is linear in the number of paths, as we show in the appendix.)
- Each loss on subflow r, decrease the window w_r by $w_r/2$.

Here RTT_r is the round trip time as measured by subflow r. We use a smoothed RTT estimator, computed similarly to TCP.

In our implementation, we compute the increase parameter only when the congestion windows grow to accommodate one more packet, rather than every ACK on every subflow.

The following subsections explain how we arrived at this design. The basic question we set out to answer is how precisely to adapt the subflow windows of a multipath TCP so as to get the maximum performance possible, subject to the constraint of co-existing gracefully with existing TCP traffic.

2.1 Fairness at shared bottlenecks

The obvious question to ask is why not just run regular TCP congestion control on each subflow? Consider the scenario in Fig. 1. If multipath TCP ran regular TCP congestion control on both paths, then the multipath flow would obtain twice as much throughput as the single path flow (assuming all RTTs are equal). This is unfair. An obvious solution is to run a weighted version of TCP on each subflow, weighted so as to take some fixed fraction of the bandwidth that regular TCP would take. The weighted TCP proposed by [5] is not suitable for weights smaller than 0.5, so instead [11] consider the following algorithm, EWTCP.

ALGORITHM: EWTCP

- For each ACK on path r, increase window w_r by a/w_r.
- For each loss on path r, decrease window w_r by $w_r/2$.

Here w_r is the window size on path r, and $a = 1/\sqrt{n}$ where n is the number of paths.

Each subflow gets window size proportional to a^2 [11]. By choosing $a = 1/\sqrt{n}$, and assuming equal RTTs, the multipath flow gets the same throughput as a regular TCP at the bottleneck link. This is an appealingly simple mechanism in that it does not require any sort of explicit shared-bottleneck detection.

2.2 Choosing efficient paths

Athough EWTCP can be fair to regular TCP traffic, it would not make very efficient use of the network. Consider the somewhat contrived scenario in Fig.2, and suppose that the three links each have capacity 12Mb/s. If each flow split its traffic evenly across its two paths[2], then each subflow would get 4Mb/s hence each flow would get 8Mb/s. But if each flow used only the one-hop shortest path, it could get 12Mb/s. (In general, however, it is not efficient to always use only shortest paths, as the simulations in §4 of data center topologies show.)

A solution has been devised in the theoretical literature on congestion control, independently by [15] and [10]. The core idea is that a multipath flow should shift all its traffic onto the least-congested path. In a situation like Fig. 2 the two-hop paths will have higher drop probability than the one-hop paths, so applying the core idea will yield the efficient allocation. Surprisingly it

[2]In this topology EWTCP wouldn't actually split its traffic evenly, since the two-hop path traverses two bottleneck links and so experiences higher congestion. In fact, as TCP's throughput is inversely proportional to the square root of loss rate, EWTCP would end up sending approximately 3.5Mb/s on the two-hop path and 5Mb/s on the single-hop path, a total of 8.5Mb/s—slightly more than with an even split, but much less than with an optimal allocation.

turns out that this can be achieved (in theory) without any need to explicitly measure congestion[3]. Consider the following algorithm, called COUPLED[4]:

ALGORITHM: COUPLED

- For each ACK on path r, increase window w_r by $1/w_{\text{total}}$.
- For each loss on path r, decrease window w_r by $w_{\text{total}}/2$.

Here w_{total} is the total window size across all subflows. We bound w_r to keep it non-negative; in our experiments we bound it to be $\geq$ 1pkt, but for the purpose of analysis it is easier to think of it as ≥ 0.

To get a feeling for the behaviour of this algorithm, we now derive an approximate throughput formula. Consider first the case that all paths have the same loss rate p. Each window w_r is made to increase on ACKs, and made to decrease on drops, and in equilibrium the increases and decreases must balance out, i.e. rate of ACKs $\times$ average increase per ACK must equal rate of drops $\times$ average decrease per drop, i.e.

$$\left(\frac{w_{\text{r}}}{\text{RTT}}(1-p)\right)\frac{1}{w_{\text{total}}} = \left(\frac{w_{\text{r}}}{\text{RTT}}p\right)\frac{w_{\text{total}}}{2}. \qquad (2)$$

Solving for w_{total} gives $w_{\text{total}} = \sqrt{2(1-p)/p} \approx \sqrt{2/p}$ (where the approximation is good if p is small). Note that when there is just one path then COUPLED reduces to regular TCP, and that the formula for w_{total} does not depend on the number of paths, hence COUPLED automatically solves the fairness problem in §2.1.

For the case that the loss rates are not all equal, let p_r be the loss rate on path r and let p_{min} be the minimum loss rate seen over all paths. The increase and decrease amounts are the same for all paths, but paths with higher p_r will see more decreases, hence the equilibrium window size on a path with $p_r > p_{\text{min}}$ is $w_r = 0$. In Fig.2, the two-hop paths go through two congested links, hence they will have higher loss rates than the one-hop paths, hence COUPLED makes the efficient choice of using only the one-hop paths.

An interesting consequence of moving traffic away from more congested paths is that loss rates across the whole network will tend to be balanced. See §3 for experiments which demonstrate this. Or consider the network shown in Fig.3, and assume all RTTs are equal.

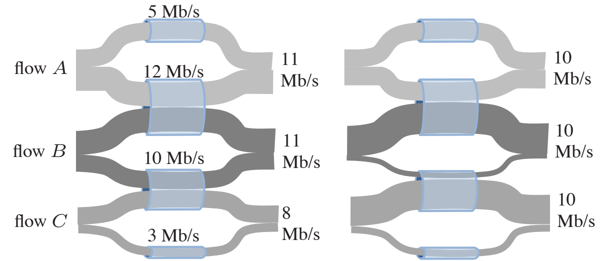


Figure 3: A scenario where EWTCP (left) does not equalize congestion or total throughput, whereas COUPLED (right) does.

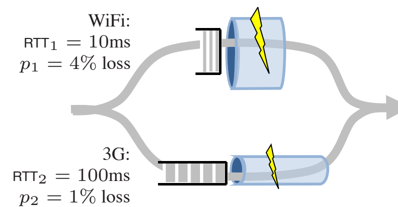


Figure 4: A scenario in which RTT and congestion mismatch can lead to low throughput.

Under EWTCP each link will be shared evenly between the subflows that use it, hence flow A gets throughputs 5 and 6 Mb/s, B gets 6 and 5 Mb/s, and C gets 5 and 3 Mb/s. Since TCP throughput is inversely related to drop probability, we deduce that the 3Mb/s link has the highest drop probability and the 12Mb/s link the lowest. For COUPLED, we can calculate the throughput on each subflow by using two facts: that a flow uses a path only if that path has the lowest loss rate p_{min} among its available paths, and that a flow's total throughput is proportional to $\sqrt{2/p_{\text{min}}}$; the only outcome consistent with these facts is for all four links to have the same loss rate, and for all flows to get the same throughput, namely 10Mb/s.

In this scenario the rule "only use a path if that path has lowest drop probability among available paths" leads to balanced congestion and balanced total throughput. In some scenarios, these may be desirable goals *per se*. Even when they are not the primary goals, they are still useful as a test: a multipath congestion control algorithm that does not balance congestion in Fig.3 is unlikely to make the efficient path choice in Fig.2.

2.3 Problems with RTT mismatch

Both EWTCP and COUPLED have problems when the RTTs are unequal. This is demonstrated by experiments in §5. To understand the issue, consider the scenario of a wireless client with two interfaces shown in Fig.4: the 3G path typically uses large buffers, resulting in long delays and low drop rates, whereas the wifi path might have smaller delays and higher drop rate. As

[3] Of course it can also be achieved by explicitly measuring congestion as in [11], but this raises tricky measurement questions.

[4] COUPLED is adapted from [15, equation (21)] and [10, equation (14)], which propose a differential equation model for a rate-based multipath version of ScalableTCP [16]. We applied the concepts behind the equations to classic window-based TCP rather than to a rate-based version of ScalableTCP, and translated the differential equations into a congestion control algorithm.

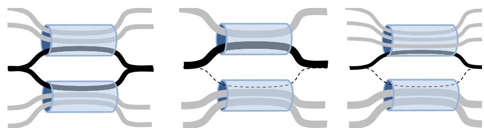


Figure 5: A scenario where multipath TCP might get 'trapped' into using a less desirable path.

a simple approximation, take the drop rates to be fixed (though in practice, e.g. in the experiments in §5, the drop rate will also depend on the sender's data rate). Also, take the throughput of a single-path TCP to be $\sqrt{2/p}/\mathsf{RTT}$ pkt/s. Then

- A single-path WiFi flow would get 707 pkt/s, and a single-path 3G flow would get 141 pkt/s.
- EWTCP is half as aggressive as single-path TCP on each path, so it will get total throughput $(707 + 141)/2 = 424$ pkt/s.
- COUPLED will send all its traffic on the less congested path, on which it will gets the same window size as single-path TCP, so it will get total throughput 141 pkt/s.[5]

Both EWTCP and COUPLED are undesirable to a user considering whether to adopt multipath TCP.

One solution is to switch from window-based control to rate-based control; the rate-based equations [15, 10] that inspired COUPLED do not suffer from RTT mismatch. But this would be a drastic change to the Internet's congestion control architecture, a change whose time has not yet come. Instead, we have a practical suggestion for window-based control, which we describe in §2.5. First though we describe another problem with COUPLED and our remedy.

2.4 Adapting to load changes

It turns out there is another pitfall with COUPLED, which shows itself even when all subflows have the same RTT. Consider the scenario in Fig. 5. Initially there are two single-path TCPs on each link, and one multipath TCP able to use both links. It should end up balancing itself evenly across the two links, since if it were uneven then one link would be more congested than the other and COUPLED would shift some of its traffic onto the less congested. Suppose now that one of the flows on the top link terminates, so the top link is less congested, hence the multipath TCP flow moves all its traffic onto the top link. But then it is 'trapped': no matter how much extra congestion there is on the top link, the the multipath TCP flow is not using the bottom link, so it gets no ACKs on the bottom link, so COUPLED is unable to increase the window size on the bottom subflow. The same problem is demonstrated in experiments in §3.

We can conclude that the simple rule "Only use the least congested paths" needs to be balanced by an opposing consideration, "Always keep sufficient traffic on other paths, as a probe, so that you can quickly discover when they improve." In fact, our implementation of COUPLED keeps window sizes $\geq$ 1pkt, so it always has some probe traffic. And the theoretical works [15, equation (11)] and [10, equation (14)] that inspired COUPLED also have a parameter that controls the amount of probing; the theory says that with infinitesimal probing one can asymptotically (after a long enough time, and with enough flows) achieve fair and efficient allocations.

But we found in experiments that if there is too little probe traffic then feedback about congestion is too infrequent for the flow to discover changes in a reasonable time. Noisy feedback (random packet drops) makes it even harder to get a quick reliable signal. As a compromise, we propose the following.

ALGORITHM: SEMICOUPLED

- For each ACK on path r, increase window w_r by a/w_{total}.
- For each loss on path r, decrease window w_r by $w_r/2$.

Here a is a constant which controls the aggressiveness, discussed below.

SEMICOUPLED tries to keep a moderate amount of traffic on each path while still having a bias in favour of the less congested paths. For example, suppose a SEMICOUPLED flow is using three paths, two with drop probability 1% and a third with drop probability 5%. We can calculate equilibrium window sizes by a balance argument similar to (2); when $1 - p_r \approx 1$ the window sizes are

$$w_r \approx \sqrt{2a}\frac{1/p_r}{\sqrt{\sum_s 1/p_s}}.$$

In three-path example, the flow will put 45% of its weight on each of the less congested path and 10% on the more congested path. This is intermediate between EWTCP (33% on each path) and COUPLED (0% on the more congested path).

To achieve fairness in scenarios like Fig.1, one can fairly simply tune the a parameter. For more complicated scenarios like Fig.4, we need a more rigorous definition of fairness, which we now propose.

2.5 Compensating for RTT mismatch

In order to reason about bias and fairness in a principled way, we propose the following two requirements for multipath congestion control:

[5] The 'proportion manager' in the multipath algorithm of [11] will also move all the traffic onto the less congested path, with the same outcome.

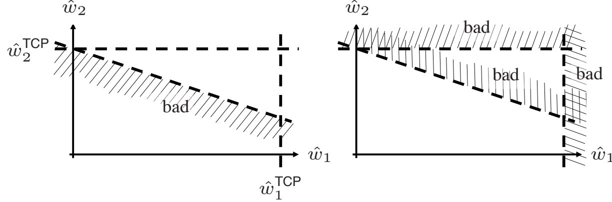


Figure 6: Fairness constraints for a two-path flow. Constraint (3) on the left, constraints (4) on the right.

- A multipath flow should give a connection at least as much throughput as it would get with single-path TCP on the best of its paths. This ensures there is an incentive for deploying multipath.
- A multipath flow should take no more capacity on any path or collection of paths than if it was a single-path TCP flow using the best of those paths. This guarantees it will not unduly harm other flows at a bottleneck link, no matter what combination of paths passes through that link.

In mathematical notation, suppose the set of available paths is R, let $\hat{w}_r$ be the equilibrium window obtained by multipath TCP on path r, and let $\hat{w}_r^{\mathsf{TCP}}$ be the equilibrium window that would be obtained by a single-path TCP experiencing path r's loss rate. We shall require

$$\sum_{r\in R} \frac{\hat{w}_r}{\mathsf{RTT}_r} \geq \max_{r\in R} \frac{\hat{w}_r^{\mathsf{TCP}}}{\mathsf{RTT}_r} \tag{3}$$

$$\sum_{r\in S} \frac{\hat{w}_r}{\mathsf{RTT}_r} \leq \max_{r\in S} \frac{\hat{w}_r^{\mathsf{TCP}}}{\mathsf{RTT}_r} \quad \text{for all } S \subseteq R. \tag{4}$$

These constraints are illustrated, for a two-path flow, in Fig.6. The left hand figure illustrates (3), namely that $(\hat{w}_1, \hat{w}_2)$ should lie on or above the diagonal line. The exact slope of the diagonal is dictated by the ratio of RTTs, and here we have chosen them so that $\hat{w}_2^{\mathsf{TCP}}/\mathsf{RTT}_2 > \hat{w}_1^{\mathsf{TCP}}/\mathsf{RTT}_1$. The right hand figure illustrates the three constraints in (4). The constraint for $S = \{\text{path}_1\}$ says to pick a point on or left of the vertical line. The constraint for $S = \{\text{path}_2\}$ says to pick a point on or below the horizontal line. The joint bottleneck constraint ($S = \{\text{path}_1, \text{path}_2\}$) says to pick a point on or below the diagonal line. Clearly the only way to satisfy both (3) & (4) is to pick some point on the diagonal, inside the box; any such point is fair. (Separately, the considerations in §2.2 say we should prefer the less-congested path, and in this figure $\hat{w}_1^{\mathsf{TCP}} > \hat{w}_2^{\mathsf{TCP}}$ hence the loss rates satisfy $p_1 < p_2$, hence we should prefer the right hand side of the diagonal line.)

The following algorithm, a modification of SEMICOUPLED, satisfies our two fairness requirements, when the flow has two paths available. EWTCP can also be fixed with a similar modification. The experiments in §5 show that the modification works.

ALGORITHM

- Each ACK on subflow r, increase the window w_r by $\min(a/w_{\text{total}}, 1/w_r)$.
- Each loss on subflow r, decrease the window w_r by $w_r/2$.

Here

$$a = \hat{w}_{\text{total}} \frac{\max_r \hat{w}_r/\mathsf{RTT}_r^2}{(\sum_r \hat{w}_r/\mathsf{RTT}_r)^2}, \tag{5}$$

w_r is the current window size on path r and $\hat{w}_r$ is the equilibrium window size on path r, and similarly for w_{total} and $\hat{w}_{\text{total}}$.

The increase and decrease rules are similar to SEMICOUPLED, so the algorithm prefers less-congested paths. The difference is that the window increase is capped at $1/w_r$, which ensures that the multipath flow can take no more capacity on either path than a single-path TCP flow would, i.e. it ensures we are inside the horizontal and vertical constraints in Fig.6.

The parameter a controls the aggressiveness. Clearly if a is very large then the two flows act like two independent flows hence the equilibrium windows will be at the top right of the box in Fig.6. On the other hand if a is very small then the flows will be stuck at the bottom left of the box. As we said, the two fairness goals require that we exactly hit the diagonal line. The question is how to find a to achieve this.

We can calculate a from the balance equations. At equilibrium, the window increases and decreases balance out on each path, hence

$$(1-p_r)\min\Big(\frac{a}{\hat{w}_{\text{total}}}, \frac{1}{\hat{w}_r}\Big) = p_r \frac{\hat{w}_r}{2}.$$

Making the approximation that p_r is small enough that $1 - p_r \approx 1$, and writing it in terms of $\hat{w}_r^{\mathsf{TCP}} = \sqrt{2/p_r}$,

$$\max\Big(\hat{w}_r, \frac{\hat{w}_{\text{total}}\hat{w}_r}{a}\Big) = \hat{w}_r^{\mathsf{TCP}}. \tag{6}$$

By simultaneously solving (3) (with the inequality replaced by equality) and (6), we arrive at (5).

Our final MPTCP algorithm, specified at the beginning of §2, is a generalization of the above algorithm to an arbitrary number of paths. The proof that it satisfies (3)–(4) is in the appendix. The formula (5) technically requires $\hat{w}_r$, the equilibrium window size, whereas in our final algorithm we have used the instantaneous window size instead. The experiments described below indicate that this does not cause problems.

Trying too hard to be fair? Our fairness goals say "take no more than a single-path TCP". At first sight this seems overly restrictive. For example, consider a single-path user with a 14.4Mb/s WiFi access link, who

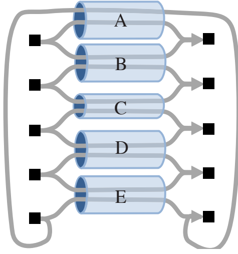


Figure 7: Torus topology. We adjust the capacity of link C, and test how well congestion is balanced.

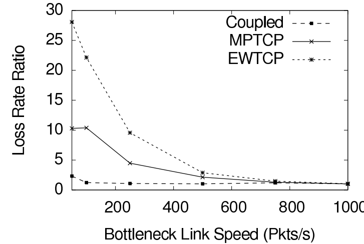


Figure 8: Effect of changing the capacity of link C on the ratio of loss rates p_C/p_A. All other links have capacity 1000pkt/s.

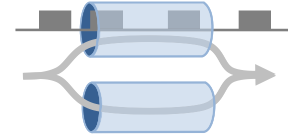


Figure 9: Bursty CBR traffic on the top link requires quick response by the multipath flow.

then adds a 2Mb/s 3G access link. Shouldn't this user now get 16.4Mb/s, and doesn't the fairness goal dictate 14.4Mb/s?

We describe tests of this scenario, and others like it, in §5. MPTCP does in fact give throughput equal to the sum of access link bandwidths, when there is no competing traffic. When there is competing traffic on the access links, the answer is different.

To understand what's going on, note that our precise fairness goals say "take no more than would be obtained by a single-path TCP *experiencing the same loss rate*". Suppose there is no competing traffic on either link, and the user only takes 14.4Mb/s. Then one or other of the two access links is underutilized, so it has no loss, and a hypothetical single-path TCP with no loss should get very high throughput, so the fairness goal allows MPTCP to increase its throughput. The system will only reach equilibrium once both access links are fully utilized. See §5 for further experimental results, including scenarios with competing traffic on the access links.

3. BALANCING CONGESTION AT A MULTIHOMED SERVER

In §3–§5 we investigate the behaviour of multipath TCP in three different scenarios: a multihomed Internet server, a data center, and a mobile client. Our aim in this paper is to produce one multipath algorithm that works robustly across a wide range of scenarios. These three scenarios will showcase all the design decisions dicussed in §2—though not all the design decisions are important in every one of the scenarios.

The first scenario is a multihomed Internet server. Multihoming of important servers has become ubiquitous over the last decade; no company reliant on network access for their business can afford to be dependent on a single upstream network. However, balancing traffic across these links is difficult, as evidenced by the hoops operators jump through using BGP techniques such as prefix splitting and AS prepending. Such techniques are coarse-grained, very slow, and a stress to the global routing system. In this section we will show that multipath transport can balance congestion, even when only a minority of flows are multipath-capable.

We will first demonstrate congestion balancing in a simple simulation, to illustrate the design discussion in §2 and to compare MPTCP to EWTCP and COUPLED. In the static scenario COUPLED is better than MPTCP is better than EWTCP, and in the dynamic scenario the order is reversed—but in each case MPTCP is close to the best, so it seems to be a reasonable compromise. We will then validate our findings with a result from an experimental testbed running our Linux implementation.

Static load balancing simulation. First we shall investigate load balancing in a stable environment of long-lived flows, testing the predictions in §2.2. Fig.7 shows a scenario with five bottleneck links arranged in a torus, each used by two multipath flows. All paths have equal RTT of 100ms, and the buffers are one bandwidth-delay product. We will adjust the capacity of link C. When the capacity of link C is reduced then it will become more congested, so the two flows using it should shift their traffic towards B and D, so those links become more congested, so there is a knock-on effect and the other flows should shift their traffic onto links A and E. With perfect balancing, the end result should be equal congestion on all links.

Fig.8 plots the imbalance in congestion as a function of the capacity of link C. When all links have equal capacity ($C = 1000$pkt/s) then congestion is of course perfectly balanced for all the algorithms. When link C is smaller, the imbalance is greater. COUPLED is very good at balancing congestion, EWTCP is bad, and MPTCP is in between. We also find that balanced congestion results in better fairness between total flow rates: when link C has capacity 100 pkt/s then Jain's fairness index is 0.99 for the flow rates with COUPLED, 0.986 for MPTCP and 0.92 for EWTCP.

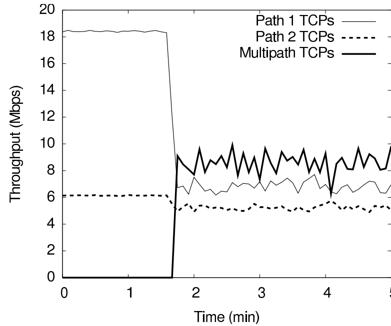


Figure 10: Server load balancing with MPTCP

Dynamic load balancing simulation. Next we illustrate the problem with dynamic load described in §2.4. We ran a simulation with two links as in Fig.9, both of capacity 100Mb/s and buffer 50 packets, and one multipath flow where each path has a 10ms RTT. On the top link there is an additional bursty CBR flow which sends at 100Mb/s for a random duration of mean 10ms, then is quiet for a random duration of mean 100ms. The multipath flow ought to use only the bottom link when the CBR flow is present, and it ought to quickly take up both links when the CBR flow is absent. We reasoned that COUPLED would do badly, and the throughputs we obtain confirm this. In Mb/s, they are

	top link	bottom link
EWTCP	85	100
MPTCP	83	99.8
COUPLED	55	99.4

We have found similar problems in a wide range of different scenarios. The exact numbers depend on how quickly congestion levels change, and in this illustration we have chosen particularly abrupt changes. One might expect similarly abrupt changes for a mobile devices when coverage on one radio interface is suddenly lost and then recovers.

Server load balancing experiment. We next give results from an experimental testbed that show our Linux implementation of MPTCP balancing congestion, validating the simulations we have just presented.

We first ran a server dual-homed with two 100Mb/s links and a number of client machines. We used dummynet to add 10ms of latency to simulate a wide-area scenario. We ran 5 client machines connecting to the server on link 1 and 15 on link 2, both using long-lived flows of Linux 2.6 NewReno TCP. The first minute of Fig.10 shows the throughput that is achieved—clearly there is more congestion on link 2. Then we started 10 multipath flows able to use both links. Perfect load balancing would require these new flows to shift completely to link 1. This is not perfectly achieved, but nonetheless multipath helps significantly to balance load, despite constituting only $1/3$ the total number of flows. The figure only shows MPTCP; COUPLED was similar and EWTCP was slightly worse as it pushed more traffic onto link 2.

Our second experiment used the same topology. On link 1 we generated Poisson arrivals of TCP flows with rate alternating between 10/s (light load) and 60/s (heavy load), with file sizes drawn from a Pareto distribution with mean 200kB. On link 2 we ran a single long-lived TCP flow. We also ran three multipath flows, one for each multipath algorithm. Their average throughputs were 61Mb/s for MPTCP, 54Mb/s for COUPLED, and 47Mb/s for EWTCP. In heavy load EWTCP did worst because it did not move as much traffic onto the less congested path. In light load COUPLED did worst because bursts of traffic on link 1 pushed it onto link 2, where it remained 'trapped' even after link 1 cleared up.

4. EFFICIENT ROUTING IN DATA CENTERS

Growth in cloud applications from companies such as Google, Microsoft and Amazon has resulted in huge data centers in which significant amounts of traffic are shifted between machines, rather than just out to the Internet. To support this, researchers have proposed new architectures with much denser interconnects than have traditionally been implemented. Two such proposals, FatTree [2] and BCube [8], are illustrated in Fig.11. The density of interconnects means that there are many possible paths between any pair of machines. The challenge is: how can we ensure that the load is efficiently distributed, no matter the traffic pattern?

One obvious benefit of any sort of multipath TCP in data centers is that it can alleviate bottlenecks at the host NICs. For example in BCube, Fig.11(b), if the core is lightly loaded and a host has a single large flow then it makes sense to use both available interfaces.

Multipath TCP is also beneficial when the network core is the bottleneck. To show this, we compared multipath TCP to single-path TCP with Equal Cost Multipath (ECMP), which we simulated by making each TCP source pick one of the shortest-hop paths at random. We ran packet-level simulations of FatTree with 128 single-interface hosts and 80 eight-port switches, and for each pair of hosts we selected 8 paths at random to use for multipath. (Our reason for choosing 8 paths is discussed below.) We also simulated BCube with 125 three-interface hosts and 25 five-port switches, and for each pair of hosts we selected 3 edge-disjoint paths according to the BCube routing algorithm, choosing the intermediate nodes at random when the algorithm needed a choice. All links were 100Mb/s.

We simulated three traffic patterns, all consisting of

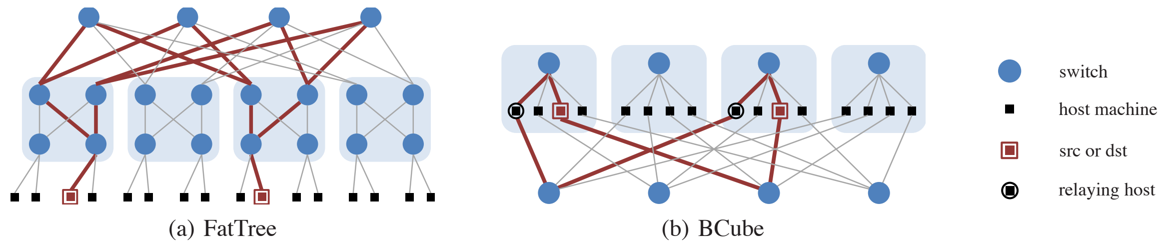


Figure 11: Two proposed data center topologies. The bold lines show multiple paths between the source and destination.

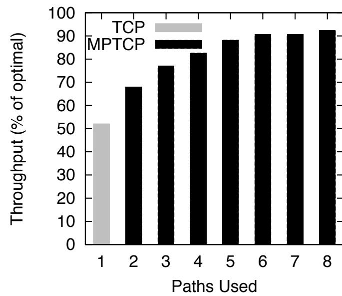


Figure 12: Multipath needs 8 paths to get good utilization in FatTree

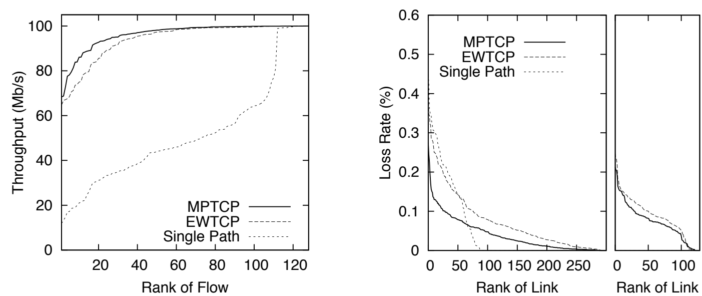


Figure 13: Distribution of throughput and loss rate, in 128-node FatTree

long-lived flows. TP1 is a random permutation where each host opens a flow to a single destination chosen uniformly at random, such that each host has a single incoming flow. For FatTree, this is the least amount of traffic that can fully utilize the network and is a good test for overall utilization. In TP2 each host opens 12 flows to 12 destinations; in FatTree the destinations are chosen at random, while in BCube they are the host's neighbours in the three levels. This mimics the locality of communication of writes in a distributed filesystem, where replicas of a block may be placed close to each other in the physical topology in order to allow higher throughput. We are using a high number of replicas as a stress-test of locality. Finally, TP3 is a sparse traffic pattern: 30% of the hosts open one flow to a single destination chosen uniformly at random.

FatTree simulations. The per-host throughputs obtained in FatTree in Mb/s, are:

	TP1	TP2	TP3
SINGLE-PATH	51	94	60
EWTCP	92	92.5	99
MPTCP	95	97	99

These figures show that for all three traffic patterns, both EWTCP and MPTCP have enough path diversity to 'find' nearly all the capacity in the network, as we can see from the fact that they get close to full utilization of the machine's 100Mb/s interface card. Fig.12 shows the throughput achieved as a function of paths used, for MPTCP under TP1—we have found that 8 is enough to get 90% utilization, in simulations across a range of traffic matrices and with thousands of hosts.

Average throughput figures do not give the full picture. Fig.13 shows the distribution of throughput on each flow, and of loss rate on each link, obtained by the three algorithms, for traffic pattern TP1. We see that MPTCP does a better job of allocating throughput fairly than EWTCP, for the reasons discussed in §2.2 and §3. Fairness matters for many datacenter distributed computations that farm processing out to many nodes and are limited by the response time of the slowest node. We also see that MPTCP does a better job of balancing congestion.

BCube simulations. The per-host throughputs obtained in BCube, in Mb/s, are:

	TP1	TP2	TP3
SINGLE-PATH	64.5	297	78
EWTCP	84	229	139
MPTCP	86.5	272	135

These throughput figures reflect three different phenomena. First, both multipath algorithms allow a host to use all three of its interfaces whereas single-path TCP can use only one, so they allow higher throughput. This is clearest in the sparse traffic pattern TP3, where the network core is underloaded. Second, BCube paths may have different hop counts, hence they are likely to traverse different numbers of bottlenecks, so some paths will be more congested than others. As discussed in §2.2, an efficient multipath algorithm should shift its

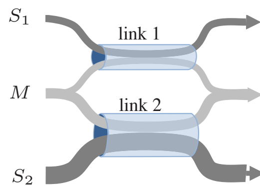


Figure 14: A multipath flow competing against two single-path flows

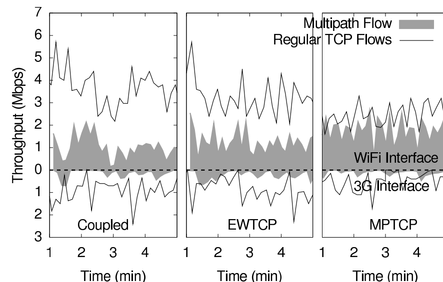


Figure 15: Multipath TCP throughput compared to single-path, where link 1 is WiFi and link 2 is 3G.

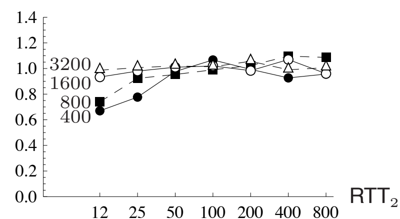


Figure 16: The ratio of flow M's throughput to the better of flow S_1 and S_2, as we vary link 2 in Fig.14.

traffic away from congestion, and EWTCP does not do this hence it tends to get worse throughput than MPTCP. This is especially clear in TP2, and not noticeable in TP3 where the core has little congestion. Third, even MPTCP does not move *all* its traffic away from the most congested path, for the reasons discussed in §2.4, so when the least-congested paths happen to all be shortest-hop paths then shortest-hop single-path TCP will do better. This is what happened in TP2. (Of course it is not always true that the least congested paths are all shortest-hop paths, so shortest-hop single-path TCP does poorly in other cases.)

In summary, MPTCP performs well across a wide range of traffic patterns. In some cases EWTCP achieves throughput as good as MPTCP, and in other cases it falls short. Even when its average throughput is as good as MPTCP it is less fair.

We have compared multipath TCP to single-path TCP, assuming that the single path is chosen at random from the shortest-hop paths available. Randomization goes some way towards balancing traffic, but it is likely to cause some congestion hotspots. An alternative solution for balancing traffic is to use a centralized scheduler which monitors large flows and solves an optimization problem to calculate good routes for them [3]. We have found that, in order to get comparable performance to MPTCP, one may need to re-run the scheduler as often as every 100ms [22] which raises serious scalability concerns. However, the exact numbers depend on the dynamics of the traffic matrix.

5. MULTIPATH WIRELESS CLIENT

Modern mobile phones and devices such as Nokia's N900 have multiple wireless interfaces such as WiFi and 3G, yet only one of them is used for data at any given time. With more and more applications requiring Internet access, from email to navigation, multipath can improve mobile users' experience by allowing simultaneous use of both interfaces. This shields the user from the inherently variable connectivity of wireless networks.

3G and WiFi have quite different link characteristics. WiFi provides much higher throughput and short RTTs, but in our tests its performance was very variable with quite high loss rates, because there was significant interference in the 2.4GHz band. 3G tends to vary over longer timescales, and we found it to be overbuffered leading to RTTs of well over a second. These differences provide a good test of the fairness goals and RTT compensation algorithm developed in §2.5. The experiments we describe here show that MPTCP gives users at least as much throughput as single-path users, and that the other multipath algorithms we have described do worse.

Single-flow experiment. Our first experiments use a laptop equipped with a 3G USB interface and a 802.11 network adapter, running our Linux implementation of MPTCP. The laptop was placed in the same room as the WiFi basestation, and 3G reception was good. The laptop did not move, so the path characteristics were reasonably static. We ran 15 tests of 20 seconds each: 5 with single-path TCP on WiFi, 5 with single-path TCP on 3G, and 5 with MPTCP. The average throughputs (with standard deviations) were 14.4 (0.2), 2.1 (0.2) and 17.3 (0.7) Mb/s respectively. As we would wish, the MPTCP user gets bandwidth roughly equal to the sum of the bandwidths of the access links.

Competing-flows experiment. We repeated the experiment, but now with competing single-path TCP flows on each of the paths, as in Fig.14. In order to showcase our algorithm for RTT compensation we repeated the experiment but replacing MPTCP first with EWTCP and then with COUPLED. The former does not have any RTT compensation built in, although the technique we used for MPTCP could be adapted. For the latter, we do not know how to build in RTT compensation.

Fig.15 shows the total throughput obtained by each of the three flows over the course of 5 minutes, one plot for each of the three multipath algorithms. The top half

of the figure shows the bandwidth achieved on the WiFi path, the bottom half shows (inverted) the throughput on the 3G path, and the range of the grey area extending into both halves shows the throughput the multipath algorithms achieved on both paths.

The figure shows that only MPTCP gives the multipath flow a fair total throughput, i.e. approximately as good as the better of the single-path competing flows, which in this case is the WiFi flow. The pictures are somewhat choppy: it seems that the WiFi basestation is underbuffered, hence the TCP sawtooth leads to peaks and troughs in throughput as measured at the receiver; it also seems the 3G link has bursts of high speed, perhaps triggered by buffer buildup. Despite these experimental vicissitudes, the long-run averages show that MPTCP does a much better job of getting fair total throughput. The long-run average throughputs in Mb/s, over 5 minutes of each setup, are:

	multipath	TCP-WiFi	TCP-3G
EWTCP	1.66	3.11	1.20
COUPLED	1.41	3.49	0.97
MPTCP	2.21	2.56	0.65

These numbers match the predictions in §2.3. COUPLED sends all its traffic on the less congested path so it often chooses to send on the 3G path and hardly uses the WiFi path. EWTCP splits its traffic so it gets the average of WiFi and 3G throughput. Only MPTCP gets close to the correct total throughput. The shortfall (2.21Mb/s for MPTCP compared to 2.56Mb/s for the best single-path TCP) may be due to difficulty in adapting to the rapidly changing 3G link speed; we continue to investigate how quickly multipath TCP should adapt to changes in congestion.

Simulations. In order to test RTT compensation across a wider range of scenarios, we simulated the topology in Fig.14 with two wired links, with capacities $C_1 = 250$pkt/s and $C_2 = 500$pkt/s, and propagation delays $\mathsf{RTT}_1 = 500$ms and $\mathsf{RTT}_2 = 50$ms. At first sight we might expect each flow to get 250pkt/s. The simulation outcome is very different: flow S_1 gets 130pkt/s, flow S_2 gets 315pkt/s and flow M gets 305pkt/s; the drop probabilities are $p_1 = 0.22\%$ and $p_2 = 0.28\%$. After some thought we realize this outcome is very nearly what we designed the algorithm to achieve. As discussed in §2.5, flow M says 'What would a single-path TCP get on path 2, based on the current loss rate? I should get at least as much!' and decides its throughput should be around 315pkt/s. It doesn't say 'What would a single-path TCP get on path 2 if I used only path 2?' which would give the answer 250pkt/s. The issue is that the multipath flow does not take account of how its actions would affect drop probabilities when it decides on its fair rate. It is difficult to see any practical alternative.

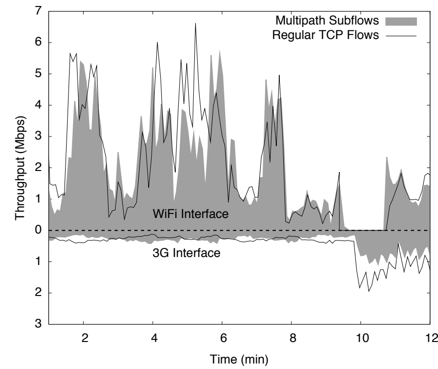


Figure 17: Throughput of multipath and regular TCP running simultaneously over 3G and WiFi. The 3G graph is shown inverted, so the total multipath throughput (the grey area) can be seen clearly.

And nonetheless, the outcome in this case is still better for both S_1 and M than if flow M used only link 1, and it is better for both S_2 and M than if flow M used only link 2.

We repeated the experiment, but with $C_1 = 400$pkt/s, $\mathsf{RTT}_1 = 100$ms, and a range of values of C_2 (shown as labels in Fig.16) and RTT_2 (the horizontal axis). Flow M aims to do as well as the better of flows S_1 and S_2. Fig.16 shows it is within a few percent of this goal in all cases except where the bandwidth delay product on link 2 is very small; in such cases there are problems due to timeouts. Over all of these scenarios, flow M always gets better throughput by using multipath than if it used just the better of the two links; the average improvement is 15%.

Mobile experiment. Having shown that our RTT compensation algorithm works in a rather testing wireless environment, we now wish to see how MPTCP performs when the client is mobile and both 3G and WiFi connectivity are intermittent. We use the same laptop and server as in the static experiment, but now the laptop user moves between floors of the building. The building has reasonable WiFi coverage on most floors but not on the staircases. 3G coverage is acceptable but is sometimes heavily congested by other users.

The experiment starts with one TCP running over the 3G interface and one over WiFi, both downloading data from an otherwise idle university server. A multipath flow then starts, using both interfaces, downloading data from the same server. Fig.17 shows the throughputs over each link (each point is a 5s average). Again, WiFi is shown above the dashed line, 3G is shown inverted below the dashed line, and the total throughput of the multipath flow can be clearly seen from the vertical range of the gray region.

During the experiment the subject moves around the building. For the first 9 minutes the 3G path has less congestion, so MPTCP would prefer to send its traffic on that route. But it also wants to get as much throughput as the higher-throughput path, in this case WiFi. The fairness algorithm prevents it from sending this much traffic on the 3G path, so as not to out-compete other single path TCPs that might be using 3G, and so the remainder is sent on WiFi. At 9 minutes the subject walks downstairs to go to a coffee machine. On the stairwell there is no WiFi coverage, but 3G coverage is better, so MPTCP adapts and takes advantage. When the subject leaves the stairwell, a new WiFi basestation is acquired, and multipath quickly takes advantage of it. This single trace shows the robustness advantage of multipath TCP, and it also shows that it does a good job of utilizing different links simultaneously without harming competing traffic on those links.

6. PROTOCOL IMPLEMENTATION

Although this paper primarily focuses on the congestion control dynamics of MPTCP, the protocol changes to TCP needed to implement multipath can be quite subtle. In particular, we must to be careful to avoid deadlock in a number of scenarios, especially relating to buffer management and flow control. In fact we discovered there is little choice in many aspects of the design. There are also many tricky issues regarding middleboxes which further constrain the design, not described here. A more complete exposition of these constraints can be found in [21], and our protocol is precisely described in the current `mptcp` draft [7].

Subflow establishment. Our implementation of MPTCP requires both client and server to have multipath extensions. A TCP option in the SYN packets of the first subflow is used to negotiate the use of multipath if both ends support it, otherwise they fall back to regular TCP behavior. After this, additional subflows can be initiated; a TCP option in the SYN packets of the new subflows allows the recipient to tie the subflow into the existing connection. We rely on multiple interfaces or multiple IP addresses to obtain different paths; we have not yet studied the question of when additional paths should be started.

Loss Detection and Stream Reassembly. Rgeular TCP uses a single sequence space for both loss detection and reassembly of the application data stream. With MPTCP, loss is a subflow issue, but the application data stream spans all subflows. To accomplish both goals using a single sequence space, the sequence space would need to be striped across the subflows. To detect loss, the receiver would then need to use selective acknowledgments and the sender would need to keep a scoreboard of which packets were sent on each subflow. Retransmitting packets on a different subflow creates an ambiguity, but the real problem is middleboxes that are unaware of MPTCP traffic. For example, the *pf*[19] firewall can re-write TCP sequence numbers to improve the randomness of the initial sequence number. If only one of the subflows passes through such a firewall, the receiver cannot reliably reconstruct the data stream.

To avoid such issues, we separated the two roles of sequence numbers. The sequence numbers and cumulative ack in the TCP header are per-subflow, allowing efficient loss detection and fast retransmission. Then to permit reliable stream reassembly, an additional data sequence number is added stating where in the application data stream the payload should be placed.

Flow Control. TCP's flow control is implemented via the combination of the receive window field and the acknowledgment field in the TCP packet header. The receive window indicates the number of bytes beyond the acknowledged sequence number that the receiver can buffer. The sender is not permitted to send more than this amount of additional data.

Multipath TCP also needs to implement flow control, although packets now arrive over multiple subflows. Two choices seem feasible:

- separate buffer pools are maintained at the receiver for each subflow, and their occupancy is signalled relative to the subflow sequence space using the receive window field.
- a single buffer pool is maintained at the receiver, and its occupancy is signalled relative to the data sequence space using the receive window field.

Unfortunately the former suffers from potential deadlock. Suppose subflow 1 stalls due to an outage, but subflow 2's receive buffer fills up. The packets received from subflow 2 cannot be passed to the application because a packet from subflow 1 is still missing, but there is no space in subflow 2's receive window to resend the packet from subflow 1 that is missing. To avoid this we use a single shared buffer; all subflows report the receive window relative to the last consecutively received data in the data sequence space.

Does the data cumulative ack then need to be explicit, or can it be inferred from subflow acks by keeping track of which data corresponds to which subflow sequence numbers?

Consider the following scenario: a receiver has sufficient buffering for two packets[6]. In accordance with the receive window, the sender sends two packets; data segment 1 is sent on subflow 1 with subflow sequence num-

[6]The same issue occurs with larger buffers

ber 10, and data segment 2 is sent on subflow 2 with subflow sequence number 20. The receiver acknowledges the packets using subflow sequence numbers only; the sender will infer which data is being acknowledged. Initially, the inferred cumulative ack is 0.

i. In the Ack for 10, the receiver acks data 1 in order, but the receiving application has not yet read the data, so relative to 1, the receive window is closed to 1 packet.
ii. In the Ack for 20, the receiver acks data 2 in order. As the application still has not read, relative to 2 the receive window is now zero.
iii. Unfortunately the acks are reordered simply because the RTT on path 2 is shorter than that on path 1, a common event. The sender receives the Ack for 20, infers that 2 has been received but 1 has not. The data cumulative ack is therefore still 0.
iv. When the ack for 10 arrives, the receiver infers that 1 and 2 have been received, so the data cumulative ack is now 2. The receive window indicated is 1 packet, relative to the inferred cumulative ack of 2. Thus the sender can send packet 3. Unfortunately, the receiver cannot buffer 3 and must drop it.

In general, the problem is that although it is possible to infer a data cumulative ack from the subflow acks, it is not possible to reliably infer the trailing edge of the receive window. The result is either missed sending opportunities or dropped packets. This is not a corner case; it will occur whenever RTTs differ so as to cause the acks to arrive in a different order from that in which they were sent.

To avoid this problem (and some others related to middleboxes) we add an explicit data acknowledgment field in addition to the subflow acknowledgment field in the TCP header.

Encoding. How should be data sequence numbers and data acknowledgments be encoded in TCP packets? Two mechanisms seemed feasible: carry them in TCP options or embed them in the payload using an SSL-like chunking mechanism. For data sequence numbers there is no compelling reason to choose one or the other, but for data acknowledgements the situation is more complex.

For the sake of concreteness, let us assume that a hypothetical payload encoding uses a chunked TLV structure, and that a data ack is contained in its own chunk, interleaved with data chunks flowing in the same direction. As data acks are now part of the data stream, they are subject to congestion control and flow control. This can lead to potential deadlock scenarios.

Consider a scenario where A's receive buffer is full because the application has not read the data, but A's application wishes to send data to B whose receive buffer is empty. This might occur for example when B is pipelining requests to A, and A now needs to send the response to an earlier request to B before reading the next request.

A sends its data, B stores it locally, and wants to send the data ACK, but can't do so: flow control imposed by A's receive window stops him. Because no data acks are received from B, A cannot free its send buffer, so this fills up and blocks the sending application on A. The connection is now deadlocked. A's application will only read when it has finished sending data to B, but it cannot do so because his send buffer is full. The send buffer can only empty when A receives an data ack from B, but B cannot send a data ack until A's application reads. This is a classic deadlock cycle.

In general, flow control of acks seems to be dangerous. Our implementation conveys data acks using TCP options to avoid this and similar issues. Given this choice, we also encode data sequence numbers in TCP options.

7. RELATED WORK

There has been a good deal of work on building multipath transport protocols [13, 27, 18, 12, 14, 6, 23, 7]. Most of this work focuses on the protocol mechanisms needed to implement multipath transmission, with key goals being robustness to long term path failures and to short term variations in conditions on the paths. The main issues are what we discussed in §6: how to split sequence numbers across paths (i.e. whether to use one sequence space for all subflows or one per subflow with an extra connection-level sequence number), how to do flow control (subflow, connection level or both), how to ack, and so forth. Our protocol design in §6 has drawn on this literature.

However, the main focus of this paper is congestion control not protocol design. In most existing proposals, the problem of shared bottlenecks (§2.1) is considered but the other issues in §2 are not. Let us highlight the congestion control characteristics of these proposals.

pTCP [12], CMT over SCTP[14] and M/TCP [23] use uncoupled congestion control on each path, and are not fair to competing single-path traffic in the general case.

mTCP [27] also performs uncoupled congestion control on each path. In an attempt to detect shared congestion at bottlenecks it computes the correlation between fast retransmit intervals on different subflows. It is not clear how robust this detector is.

R-MTP [18] targets wireless links: it probes the bandwidth available periodically for each subflow and adjusts the rates accordingly. To detect congestion it uses packet interarrival times and jitter, and infers mounting congestion when it observes increased jitter. This only works when wireless links are the bottleneck.

The work in [11] is based on using EWTCP with different weights on each path, and adapting the weights to

achieve the outcomes described in §2.1–§2.2. It does not address the problems identified in §2.3–§2.5, and in particular it has problems coping with heterogenous RTTs.

Network layer multipath. ECMP[25] achieves load balancing at the flow level, without the involvement of end-systems. It sends all packets from a given flow along the same route in order that end-systems should not see any packet re-ordering. ECMP and multipath TCP complement each other. Multipath TCP can use ECMP to get different paths through the network without having multihomed endpoints. Different subflows of the same multipath connection will have different five-tuples (at least one port will differ) and will likely hash onto a different path with ECMP. This interaction can be readily used in data centers, where multiple paths are available and ECMP is widely used.

Horizon [20] is a system for load balancing at the network layer, for wireless mesh networks. Horizon network nodes maintain congestion state and estimated delay for each possible path towards the destination; hop-by-hop backpressure is applied to achieve near-optimal throughput, and the delay estimates let it avoid re-ordering. Theoretical work suggests that inefficient outcomes may arise when both the end-systems and the network participate in balancing traffic [1].

Application layer multipath. BitTorrent [4] is an example of application layer multipath. Different chunks of the same file are downloaded from different peers to increase throughput. BitTorrent works at chunk granularity, and only optimizes for throughput, downloading more chunks from faster servers. Essentially BitTorrent is behaving in a similar way to uncoupled multipath congestion control, albeit with the paths having different endpoints. While uncoupled congestion control does not balance flow rates, it nevertheless achieves some degree of load balancing when we take into account flow sizes [17, 26], by virtue of the fact that the less congested subflow gets higher throughput and therefore fewer bytes are put on the more congested subflow.

8. CONCLUSIONS & FUTURE WORK

We have demonstrated a working multipath congestion control algorithm. It brings immediate practical benefits: in §5 we saw it seamlessly balance traffic over 3G and WiFi radio links, as signal strength faded in and out. It is safe to use: the fairness mechanism from §2.5 ensures that it does not harm other traffic, and that there is always an incentive to turn it on because its aggregate throughput is at least as good as would be achieved on the best of its available paths. It should be beneficial to the operation of the Internet, since it selects efficient paths and balances congestion, as described in §2.2 and demonstrated in §3, at least in so far as it can given topological constraints and the requirements of fairness.

We believe our multipath congestion control algorithm is safe to deploy, either as part of the IETF's efforts to standardize Multipath TCP[7] or with SCTP, and it will perform well. This is timely, as the rise of multipath-capable smart phones and similar devices has made it crucial to find a good way to use multiple interfaces more effectively. Currently such devices use heuristics to periodically choose the best interface, terminating existing connections and re-establishing new ones each time a switch is made. Combined with a transport protocol such as Multipath TCP or SCTP, our congestion control mechanism avoids the need to make such binary decisions, but instead allows continuous and rapid rebalancing on short timescales as wireless conditions change.

Our congestion control scheme is designed to be compatible with existing TCP behavior. However, existing TCP has well-known limitations when coping with long high-speed paths. To this end, Microsoft incorporate Compound TCP[24] in Vista and Windows 7, although it is not enabled by default, and recent Linux kernels use Cubic TCP[9]. We believe that Compound TCP should be a very good match for our congestion control algorithm. Compound TCP kicks in when a link is underutilized to rapidly fill the pipe, but it falls back to NewReno-like behavior once a queue starts to build. Such a delay-based mechanism would be complementary to the work described in this paper, but would further improve a multipath TCP's ability to switch to a previously congested path that suddenly has spare capacity. We intend to investigate this in future work.

9. REFERENCES

[1] D. Acemoglu, R. Johari, and A. Ozdaglar. Partially optimal routing. *IEEE Journal of selected areas in communications*, 2007.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

[3] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. NSDI*, 2010.

[4] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on econonomics of peer-to-peer systems*, 2003.

[5] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *CCR*, 1998.

[6] Y. Dong, D. Wang, N. Pissinou, and J. Wang. Multi-path load balancing in transport layer. In *Proc. 3rd EuroNGI Conference on Next Generation Internet Networks*, 2007.

[7] A. Ford, C. Raiciu, and M. Handley. TCP extensions for multipath operation with multiple addresses, Oct 2010. IETF draft (work in progress).

[8] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: a high performance,

server-centric network architecture for modular data centers. In *Proc. SIGCOMM*, 2009.

[9] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5), 2008.

[10] H. Han, S. Shakkottai, C. V. Hollot, R. Srikant, and D. Towsley. Multi-path TCP: a joint congestion control and routing scheme to exploit path diversity in the Internet. *IEEE/ACM Trans. Networking*, 14(6), 2006.

[11] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda. Multipath Congestion Control for Shared Bottleneck. In *Proc. PFLDNeT workshop*, May 2009.

[12] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proc. MobiCom '02*, pages 83–94, New York, NY, USA, 2002. ACM.

[13] C. Huitema. Multi-homed TCP. Internet draft, IETF, 1995.

[14] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Trans. Netw.*, 14(5):951–964, 2006.

[15] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *CCR*, 35(2), Apr. 2005.

[16] T. Kelly. Scalable TCP: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, 2003.

[17] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proc. IEEE Infocom*, May 2007. Also appeared in proceedings of IEEE ICASSP 2007.

[18] L. Magalhaes and R. Kravets. Transport level mechanisms for bandwidth aggregation on mobile hosts. *ICNP*, page 0165, 2001.

[19] *PF: the OpenBSD Packet Filter*. OpenBSD 4.7, `www.openbsd.org/faq/pf`, retrieved Sep 2010.

[20] B. Radunović, C. Gkantsidis, D. Gunawardena, and P. Key. Horizon: balancing TCP over multiple paths in wireless mesh network. In *Proc. MobiCom '08*, 2008.

[21] C. Raiciu, M. Handley, and A. Ford. Multipath TCP design decisions. Work in progress, `www.cs.ucl.ac.uk/staff/C.Raiciu/files/mtcp-design.pdf`, 2009.

[22] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data center networking with multipath TCP. In *Hotnets*, 2010.

[23] K. Rojviboonchai and H. Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE Trans. Communications*, 2004.

[24] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP approach for high-speed and long distance networks. In *Proc. IEEE INFOCOM 2006*, pages 1–12, April 2006.

[25] D. Thaler and C. Hopps. Multipath Issues in Unicast and Multicast Next-Hop Selection. RFC 2991 (Informational), Nov. 2000.

[26] B. Wang, W. Wei, J. Kurose, D. Towsley, K. R. Pattipati, Z. Guo, and Z. Peng. Application-layer multipath data transfer via TCP: schemes and performance tradeoffs. *Performance Evaluation*, 64(9–12), 2007.

[27] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proc USENIX '04*, 2004.

Appendix

We now prove that the equilibrium window sizes of MPTCP satisfy the fairness goals in §2.5. The rough intuition is that if we use SEMICOUPLED from §2.4, and additionally ensure (4), then the set of bottlenecked paths increases as a increases. The proof involves identifying the order in which paths become bottlenecked, to permit an analysis similar to §2.5.

First define

$$i(S) = \frac{\max_{r\in S} \sqrt{\hat{w}_r}/\mathsf{RTT}_r}{\sum_{r\in S} \hat{w}_r/\mathsf{RTT}_r}$$

and assume for convenience that the window sizes are kept in the order

$$\frac{\sqrt{\hat{w}_1}}{\mathsf{RTT}_1} \le \frac{\sqrt{\hat{w}_2}}{\mathsf{RTT}_2} \le \cdots \frac{\sqrt{\hat{w}_n}}{\mathsf{RTT}_n}.$$

Note that with this ordering, the equilibrium window increase (1) reduces to

$$\min_{S\subseteq R: r\in S} \frac{\hat{w}_{\max(S)}/\mathsf{RTT}^2_{\max(S)}}{\left(\sum_{s\in S} w_s/\mathsf{RTT}_s\right)^2} = \min_{r\le u\le n} \frac{\hat{w}_u/\mathsf{RTT}^2_u}{\left(\sum_{t\le u} w_t/\mathsf{RTT}_t\right)^2}$$

i.e. it can be computed with a linear search not a combinatorial search.

At equilibrium, assuming drop probabilities are small so $1 - p_r \approx 1$, the window sizes satisfy the balance equations

$$\min_{S: r\in S} i(S)^2 = p_r \hat{w}_r/2 \quad \text{for each } r \in R.$$

Rearranging this, and writing it in terms of $\hat{w}_r = \sqrt{2/p_r}$,

$$\hat{w}^{\mathsf{TCP}}_r = \sqrt{\hat{w}_r} \max_{S: r\in S} 1/i(S). \tag{7}$$

Now take any $T \subseteq R$. Rearranging the definition of $i(T)$, and applying some simple algebra, and substituting in (7),

$$\sum_{r\in T} \frac{\hat{w}_r}{\mathsf{RTT}_r} = \max_{r\in T} \frac{1}{\mathsf{RTT}_r} \sqrt{\hat{w}_r}/i(T) \le \max_{r\in T} \frac{1}{\mathsf{RTT}_r} \sqrt{\hat{w}_r} \max_{S: r\in S} 1/i(S) = \max_{r\in T} \frac{\hat{w}^{\mathsf{TCP}}_r}{\mathsf{RTT}_r}.$$

Since T was arbitrary, this proves we satisfy (4).

To prove (3), applying (7) at $r = n$ in conjunction with the ordering on window sizes, we get

$$\frac{\hat{w}^{\mathsf{TCP}}_n}{\mathsf{RTT}_n} = \sum_r \frac{\hat{w}_r}{\mathsf{RTT}_r}.$$

One can also show that for all r, $\hat{w}^{\mathsf{TCP}}_r/\mathsf{RTT}_r \le \hat{w}^{\mathsf{TCP}}_n/\mathsf{RTT}_n$; the proof is by induction on r starting at $r = n$, and is omitted. These two facts imply (3).

CIEL: a universal execution engine for distributed data-flow computing

Derek G. Murray Malte Schwarzkopf Christopher Smowton
Steven Smith Anil Madhavapeddy Steven Hand
University of Cambridge Computer Laboratory

Abstract

This paper introduces CIEL, a universal execution engine for distributed data-flow programs. Like previous execution engines, CIEL masks the complexity of distributed programming. Unlike those systems, a CIEL job can make data-dependent control-flow decisions, which enables it to compute iterative and recursive algorithms.

We have also developed Skywriting, a Turing-complete scripting language that runs directly on CIEL. The execution engine provides transparent fault tolerance and distribution to Skywriting scripts and high-performance code written in other programming languages. We have deployed CIEL on a cloud computing platform, and demonstrate that it achieves scalable performance for both iterative and non-iterative algorithms.

1 Introduction

Many organisations have an increasing need to process large data sets, and a cluster of commodity machines on which to process them. *Distributed execution engines*—such as MapReduce [18] and Dryad [26]—have become popular systems for exploiting such clusters. These systems expose a simple programming model, and automatically handle the difficult aspects of distributed computing: fault tolerance, scheduling, synchronisation and communication. MapReduce and Dryad can be used to implement a wide range of algorithms [3, 39], but they are awkward or inefficient for others [12, 21, 25, 28, 34]. The problems typically arise with *iterative* algorithms, which underlie many machine-learning and optimisation problems, but require a more *expressive* programming model and a more powerful execution engine. To address these limitations, and extend the benefits of distributed execution engines to a wider range of applications, we have developed Skywriting and CIEL.

Skywriting is a scripting language that allows the straightforward expression of iterative and recursive task-parallel algorithms using imperative and functional language syntax [31]. Skywriting scripts run on CIEL, an execution engine that provides a *universal* execution model for distributed data-flow. Like previous systems, CIEL coordinates the distributed execution of a set of data-parallel tasks arranged according to a data-flow DAG, and hence benefits from transparent scaling and fault tolerance. However CIEL extends previous models by *dynamically* building the DAG as tasks execute. As we will show, this conceptually simple extension—allowing tasks to create further tasks—enables CIEL to support data-dependent iterative or recursive algorithms. We present the high-level architecture of CIEL in Section 3, and explain how Skywriting maps onto CIEL's primitives in Section 4.

Our implementation incorporates several additional features, described in Section 5. Like existing systems, CIEL provides transparent fault tolerance for worker nodes. Moreover, CIEL can tolerate failures of the cluster master and the client program. To improve resource utilisation and reduce execution latency, CIEL can memoise the results of tasks. Finally, CIEL supports the streaming of data between concurrently-executing tasks.

We have implemented a variety of applications in Skywriting, including MapReduce-style (grep, wordcount), iterative (k-means, PageRank) and dynamic-programming (Smith-Waterman, option pricing) algorithms. In Section 6 we evaluate the performance of some of these applications when run on a CIEL cluster.

2 Motivation

Several researchers have identified limitations in the MapReduce and Dryad programming models. These systems were originally developed for batch-oriented jobs, namely large-scale text mining for information retrieval [18, 26]. They are designed to maximise throughput, rather than minimise individual job latency. This is especially noticeable in iterative computations, for which

Feature	MapReduce [2, 18]	Dryad [26]	Pregel [28]	Iterative MR [12, 21]	Piccolo [34]	CIEL
Dynamic control flow	✗	✗	✓	✓	✓	✓
Task dependencies	Fixed (2-stage)	Fixed (DAG)	Fixed (BSP)	Fixed (2-stage)	Fixed (1-stage)	Dynamic
Fault tolerance	Transparent	Transparent	Transparent	✗	Checkpoint	Transparent
Data locality	✓	✓	✓	✓	✓	✓
Transparent scaling	✓	✓	✓	✓	✗	✓

Figure 1: Analysis of the features provided by existing distributed execution engines.

multiple jobs are chained together and the job latency is multiplied [12, 21, 25, 28, 34].

Nevertheless, MapReduce—in particular its open-source implementation, Hadoop [2]—remains a popular platform for parallel iterative computations with large inputs. For example, the Apache Mahout machine learning library uses Hadoop as its execution engine [3]. Several of the Mahout algorithms—such as k-means clustering and singular value decomposition—are iterative, comprising a data-parallel kernel inside a while-not-converged loop. Mahout uses a *driver program* that submits multiple jobs to Hadoop and performs convergence testing at the client. However, since the driver program executes logically (and often physically) outside the Hadoop cluster, each iteration incurs job-submission overhead, and the driver program does not benefit from transparent fault tolerance. These problems are not unique to Hadoop, but are shared with both the original version of MapReduce [18] and Dryad [26].

The computational power of a distributed execution engine is determined by the data flow that it can express. In MapReduce, the data flow is limited to a bipartite graph parameterised by the number of map and reduce tasks; Dryad allows data flow to follow a more general directed acyclic graph (DAG), but it must be fully specified before starting the job. In general, to support iterative or recursive algorithms within a single job, we need *data-dependent control flow*—i.e. the ability to create more work dynamically, based on the results of previous computations. At the same time, we wish to retain the existing benefits of task-level parallelism: transparent fault tolerance, locality-based scheduling and transparent scaling. In Figure 1, we analyse a range of existing systems in terms of these objectives.

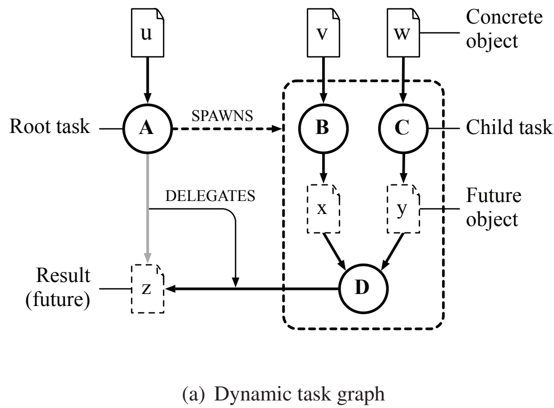
MapReduce and Dryad already support transparent fault tolerance, locality-based scheduling and transparent scaling [18, 26]. In addition, Dryad supports arbitrary task dependencies, which enables it to execute a larger class of computations than MapReduce. However, neither supports data-dependent control flow, so the work in each computation must be statically pre-determined.

A variety of systems provide data-dependent control flow but sacrifice other functionality. Google's Pregel is the largest-scale example of a distributed execution engine with support for control flow [28]. Pregel is a Bulk Synchronous Parallel (BSP) system designed for executing graph algorithms (such as PageRank), and Pregel computations are divided into "supersteps", during which a "vertex method" is executed for each vertex in the graph. Crucially, each vertex can vote to terminate the computation, and the computation terminates when all vertices vote to terminate. Like a simple MapReduce job, however, a Pregel computation only operates on a single data set, and the programming model does not support the composition of multiple computations.

Two recent systems add iteration capabilities to MapReduce. CGL-MapReduce is a new implementation of MapReduce that caches static (loop-invariant) data in RAM across several MapReduce jobs [21]. HaLoop extends Hadoop with the ability to evaluate a convergence function on reduce outputs [12]. Neither system provides fault tolerance across multiple iterations, and neither can support Dryad-style task dependency graphs.

Finally, Piccolo is a new programming model for data-parallel programming that uses a partitioned in-memory key-value table to replace the reduce phase of MapReduce [34]. A Piccolo program is divided into "kernel" functions, which are applied to table partitions in parallel, and typically write key-value pairs into one or more other tables. A "control" function coordinates the kernel functions, and it may perform arbitrary data-dependent control flow. Piccolo supports user-assisted checkpointing (based on the Chandy-Lamport algorithm), and is limited to fixed cluster membership. If a single machine fails, the entire computation must be restarted from a checkpoint with the same number of machines.

We believe that CIEL is the first system to support all five goals in Figure 1, but it is not a panacea. CIEL is designed for coarse-grained parallelism across large data sets, as are MapReduce and Dryad. For fine-grained tasks, a work-stealing scheme is more appropriate [11]. Where the entire data set can fit in RAM, Piccolo may be more efficient, because it can avoid writing to disk. Ultimately, achieving the highest performance requires significant developer effort, using a low-level technique such as explicit message passing [30].



(a) Dynamic task graph

Task ID	Dependencies	Expected outputs
A	{ u }	~~z~~
B	{ v }	x
C	{ w }	y
D	{ x, y }	z

Object ID	Produced by	Locations
u	–	{ `host19`, `host85` }
v	–	{ `host21`, `host23` }
w	–	{ `host22`, `host57` }
x	**B**	∅
y	**C**	∅
z	~~**A**~~ **D**	∅

(b) Task and object tables

Figure 2: A CIEL job is represented by a dynamic task graph, which contains tasks and objects (§3.1). In this example, root task **A** spawns tasks **B**, **C** and **D**, and delegates the production of its result to **D**. Internally, CIEL uses task and object tables to represent the graph (§3.3).

3 CIEL

CIEL is a distributed execution engine that can execute programs with arbitrary data-dependent control flow. In this section, we first describe the core abstraction that CIEL supports: the *dynamic task graph* (§3.1). We then describe how CIEL executes a job that is represented as a dynamic task graph (§3.2). Finally, we describe the concrete architecture of a CIEL cluster that is used for distributed data-flow computing (§3.3).

3.1 Dynamic task graphs

In this subsection, we define the three CIEL primitives—objects, references and tasks—and explain how they are related in a dynamic task graph (Figure 2).

CIEL is a data-centric execution engine: the goal of a CIEL job is to produce one or more output **objects**. An object is an unstructured, finite-length sequence of bytes. Every object has a unique *name*: if two objects exist with the same name, they must have the same contents. To simplify consistency and replication, an object is immutable once it has been written, but it is sometimes possible to append to an object (§5.3).

It is helpful to be able to describe an object without possessing its full contents; CIEL uses **references** for this purpose. A reference comprises a name and a set of locations (e.g. hostname-port pairs) where the object with that name is stored. The set of locations may be empty: in that case, the reference is a *future reference* to an object that has not yet been produced. Otherwise, it is a *concrete reference*, which may be consumed.

A CIEL job makes progress by executing **tasks**. A task is a non-blocking atomic computation that executes completely on a single machine. A task has one or more *dependencies*, which are represented by references, and the task becomes runnable when all of its dependencies become concrete. The dependencies include a special object that specifies the behaviour of the task (such as an executable binary or a Java class) and may impose some structure over the other dependencies. To simplify fault tolerance (§5.2), CIEL requires that all tasks compute a deterministic function of their dependencies. A task also has one or more *expected outputs*, which are the names of objects that the task will either create or delegate another task to create.

Tasks can have two externally-observable behaviours. First, a task can **publish** one or more objects, by creating a concrete reference for those objects. In particular, the task can publish objects for its expected outputs, which may cause other tasks to become runnable if they depend on those outputs. To support data-dependent control flow, however, a task may also **spawn** new tasks that perform additional computation. CIEL enforces the following conditions on task behaviour:

1. For each of its expected outputs, a task must either publish a concrete reference, or spawn a child task with that name as an expected output. This ensures that, as long as the children eventually terminate, any task that depends on the parent's output will eventually become runnable.

2. A child task must only depend on concrete references (i.e. objects that already exist) or future references to the outputs of tasks that have already been spawned (i.e. objects that are already expected to be published). This prevents deadlock, as a cycle cannot form in the dependency graph.

The **dynamic task graph** stores the relation between tasks and objects. An edge from an object to a task means

that the task depends on that object. An edge from a task to an object means that the task is expected to output the object. As a job runs, new tasks are added to the dynamic task graph, and the edges are rewritten when a newly-spawned task is expected to produce an object.

The dynamic task graph provides low-level data-dependent control flow that resembles tail recursion: a task either produces its output (analogous to returning a value) or spawns a new task to produce that output (analogous to a tail call). It also provides facilities for data-parallelism, since independent tasks can be dispatched in parallel. However, we do not expect programmers to construct dynamic task graphs manually, and instead we provide the Skywriting script language for generating these graphs programmatically (§4).

3.2 Evaluating objects

Given a dynamic task graph, the role of CIEL is to evaluate one or more objects that correspond to the job outputs. Indeed, a CIEL job can be specified as a single *root task* that has only concrete dependencies, and an expected output that names the final result of the computation. This leads to two natural strategies, which are variants of topological sorting:

Eager evaluation. Since the task dependencies form a DAG, at least one task must have only concrete dependencies. Start by executing the tasks with only concrete dependencies; subsequently execute tasks when all of their dependencies become concrete.

Lazy evaluation. Seek to evaluate the expected output of the root task. To evaluate an object, identify the task, T, that is expected to produce the object. If T has only concrete dependencies, execute it immediately; otherwise, block T and recursively evaluate all of its unfulfilled dependencies using the same procedure. When the inputs of a blocked task become concrete, execute it. When the production of a required object is delegated to a spawned task, re-evaluate that object.

When we first developed CIEL, we experimented with both strategies, but switched exclusively to lazy evaluation since it more naturally supports the fault-tolerance and memoisation features that we describe in §5.

3.3 System architecture

Figure 3 shows the architecture of a CIEL cluster. A single *master* coordinates the end-to-end execution of jobs, and several *workers* execute individual tasks.

The master maintains the current state of the dynamic task graph in the *object table* and *task table* (Figure 2(b)).

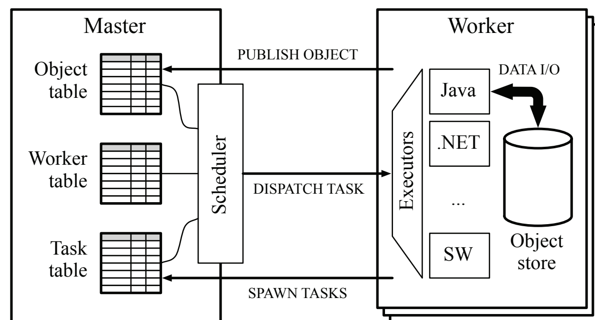


Figure 3: A CIEL cluster has a single master and many workers. The master dispatches tasks to the workers for execution. After a task completes, the worker publishes a set of objects and may spawn further tasks.

Each row in the object table contains the latest reference for that object, including its locations (if any), and a pointer to the task that is expected to produce it (if any: an object will not have a task pointer if it is loaded into the cluster by an external tool). Each row in the task table corresponds to a spawned task, and contains pointers to the references on which the task depends.

The master *scheduler* is responsible for making progress in a CIEL computation: it lazily evaluates output objects and pairs runnable tasks with idle workers. Since task inputs and outputs may be very large (on the order of gigabytes per task), all bulk data is stored on the workers themselves, and the master handles references. The master uses a multiple-queue-based scheduler (derived from Hadoop [2]) to dispatch tasks to the worker nearest the data. If a worker needs to fetch a remote object, it reads the object directly from another worker.

The workers execute tasks and store objects. At startup, a worker registers with the master, and periodically sends a heartbeat to demonstrate its continued availability. When a task is dispatched to a worker, the appropriate *executor* is invoked. An executor is a generic component that prepares input data for consumption and invokes some computation on it, typically by executing an external process. We have implemented simple executors for Java, .NET, shell-based and native code, as well as a more complex executor for Skywriting (§4).

Assuming that a worker executes a task successfully, it will reply to the master with the set of references that it wishes to publish, and a list of task descriptors for any new tasks that it wishes to spawn. The master will then update the object table and task table, and re-evaluate the set of tasks now runnable.

In addition to the master and workers, there will be one or more *clients* (not shown). A client's role is minimal: it submits a job to the master, and either polls the master to discover the job status or blocks until the job completes.

```
function process_chunk(chunk, prev_result) {
  // Execute native code for chunk processing.
  // Returns a reference to a partial result.
  return spawn_exec(...);
}

function is_converged(curr_result, prev_result) {
  // Execute native code for convergence test.
  // Returns a reference to a boolean.
  return spawn_exec(...)[0];
}

input_data = [ref("ciel://host137/chunk0"),
              ref("ciel://host223/chunk1"),
              ...];
curr = ...; // Initial guess at the result.

do {
  prev = curr;
  curr = [];
  for (chunk in input_data) {
    curr += process_chunk(chunk, prev);
  }
} while (!*is_converged(curr, prev));

return curr;
```

Figure 4: Iterative computation implemented in Skywriting. input_data is a list of n input chunks, and curr is initialised to a list of n partial results.

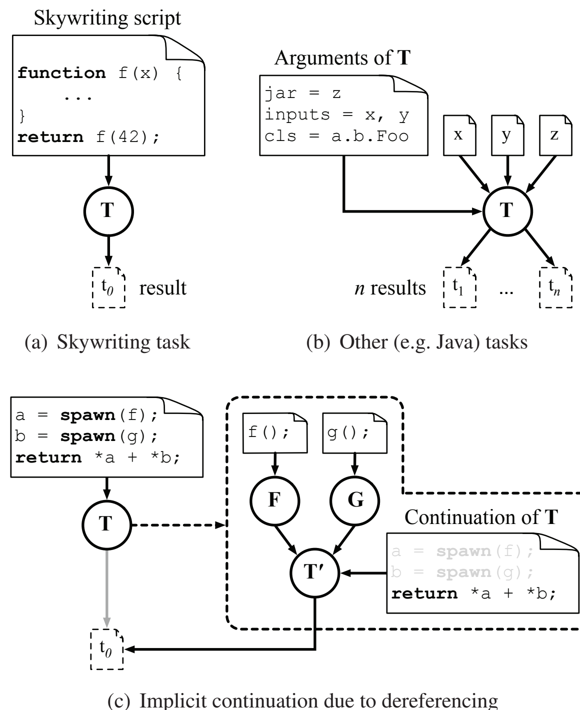


Figure 5: Task creation in Skywriting. Tasks can be created using (a) spawn(), (b) spawn_exec() and (c) the dereference (*) operator.

A job submission message contains a root task, which must have only concrete dependencies. The master adds the root task to the task table, and starts the job by lazily evaluating its output (§3.2).

Note that CIEL currently uses a single (active) master for simplicity. Despite this, our implementation can recover from master failure (§5.2), and it did not cause a performance bottleneck during our evaluation (§6). Nonetheless, if it became a concern in future, it would be possible to partition the master state—i.e. the task table and object table—between several hosts, while retaining the functionality of a single logical master.

4 Skywriting

Skywriting is a language for expressing task-level parallelism that runs on top of CIEL. Skywriting is Turing-complete, and can express arbitrary data-dependent control flow using constructs such as while loops and recursive functions. Figure 4 shows an example Skywriting script that computes an iterative algorithm; we use a similar structure in the k-means experiment (§6.2).

We introduced Skywriting in a previous paper [31], but briefly restate the key features here:

- ref(url) returns a *reference* to the data stored at the given URL. The function supports common URL schemes, and the custom ciel scheme, which accesses entries in the CIEL object table. If the URL is external, CIEL downloads the data into the cluster as an object, and assigns a name for the object.
- spawn(f, [arg, ...]) spawns a parallel task to evaluate f(arg, ...). Skywriting functions are *pure*: functions cannot have side-effects, and all arguments are passed by value. The return value is a reference to the result of f(arg, ...).
- exec(executor, args, n) synchronously runs the named executor with the given args. The executor will produce n outputs. The return value is a list of n references to those outputs.
- spawn_exec(executor, args, n) spawns a parallel task to run the named executor with the given args. As with exec(), the return value is a list of n references to those outputs.
- The dereference (unary-*) operator can be applied to any reference; it loads the referenced data into the Skywriting execution context, and evaluates to the resulting data structure.

In the following, we describe how Skywriting maps on to CIEL primitives. We describe how tasks are created (§4.1), how references are used to facilitate data-dependent control flow (§4.2), and the relationship between Skywriting and other frameworks (§4.3).

4.1 Creating tasks

The distinctive feature of Skywriting is its ability to spawn new tasks in the middle of executing a job. The language provides two explicit mechanisms for spawning new tasks (the `spawn()` and `spawn_exec()` functions) and one implicit mechanism (the $*$-operator). Figure 5 summarises these mechanisms.

The `spawn()` function creates a new task to run the given Skywriting function. To do this, the Skywriting runtime first creates a data object that contains the new task's environment, including the text of the function to be executed and the values of any arguments passed to the function. This object is called a Skywriting *continuation*, because it encapsulates the state of a computation. The runtime then creates a task descriptor for the new task, which includes a dependency on the new continuation. Finally, it assigns a reference for the task result, which it returns to the calling script. Figure 5(a) shows the structure of the created task.

The `spawn_exec()` function is a lower-level task-creation mechanism that allows the caller to invoke code written in a different language. Typically, this function is not called directly, but rather through a wrapper for the relevant executor (e.g. the built-in `java()` library function). When `spawn_exec()` is called, the runtime serialises the arguments into a data object and creates a task that depends on that object (Figure 5(b)). If the arguments to `spawn_exec()` include references, the runtime adds those references to the new task's dependencies, to ensure that CIEL will not schedule the task until all of its arguments are available. Again, the runtime creates references for the task outputs, and returns them to the calling script. We discuss how names are chosen in §5.1.

If the task attempts to dereference an object that has not yet been created—for example, the result of a call to `spawn()`—the current task must block. However, CIEL tasks are *non-blocking*: all synchronisation (and data-flow) must be made explicit in the dynamic task graph (§3.1). To resolve this contradiction, the runtime implicitly creates a *continuation task* that depends on the dereferenced object and the current continuation (i.e. the current Skywriting execution stack). The new task therefore will only run when the dereferenced object has been produced, which provides the necessary synchronisation. Figure 5(c) shows the dependency graph that results when a task dereferences the result of `spawn()`.

A task terminates when it reaches a `return` statement (or it blocks on a future reference). A Skywriting task has a single output, which is the value of the expression in the `return` statement. On termination, the runtime stores the output in the local object store, publishes a concrete reference to the object, and sends a list of spawned tasks to the master, in order of creation.

Skywriting ensures that the dynamic task graph remains acyclic. A task's dependencies are fixed when the task-creation function is evaluated, which means that they can only include references that are stored in the local Skywriting scope before evaluating the function. Therefore, a task cannot depend on itself or any of its descendants. Note that the results of `spawn()` and `spawn_exec()` are first-class *futures* [24]: a Skywriting task can pass the references in its return value or in a subsequent call to the task-creation functions. This enables a script to create arbitrary acyclic dependency graphs, such as the MapReduce dependency graph (§4.3).

4.2 Data-dependent control flow

Skywriting is designed to coordinate *data-centric* computations, which means that the objects in the computation can be divided into two spaces:

Data space. Contains large data objects that may be up to several gigabytes in size.

Coordination space. Contains small objects—such as integers, booleans, strings, lists and dictionaries—that determine the control flow.

In general, objects in the data space are processed by programs written in compiled languages, to achieve better I/O or computational performance than Skywriting can provide. In existing distributed execution engines (such as MapReduce and Dryad), the data space and coordination space are disjoint, which prevents these systems from supporting data-dependent control flow.

To support data-dependent control flow, data must be able to pass from the data space into the coordination space, so that it can help to determine the control flow. In Skywriting, the $*$-operator transforms a reference to a (data space) object into a (coordination space) value. The producing task, which may be run by any executor, must write the referenced object in a format that Skywriting can recognise; we use JavaScript Object Notation (JSON) for this purpose [4]. This serialisation format is only used for references that are passed to Skywriting, and the majority of executors use the appropriate binary format for their data.

4.3 Other languages and frameworks

Systems like MapReduce have become popular, at least in part, because of their simple interface: a developer can specify a whole distributed computation with just a pair of `map()` and `reduce()` functions. To demonstrate that Skywriting approaches this level of simplicity, Figure 6 shows an implementation of the MapReduce execution model, taken from the Skywriting standard library.

```
function apply(f, list) {
  outputs = [];
  for (i in range(len(list))) {
    outputs[i] = f(list[i]);
  }
  return outputs;
}

function shuffle(inputs, num_outputs) {
  outputs = [];
  for (i in range(num_outputs)) {
    outputs[i] = [];
    for (j in range(len(inputs))) {
      outputs[i][j] = inputs[j][i];
    }
  }
  return outputs;
}

function mapreduce(inputs, mapper, reducer, r) {
  map_outputs = apply(mapper, inputs);
  reduce_inputs = shuffle(map_outputs, r);
  reduce_outputs = apply(reducer, reduce_inputs);
  return reduce_outputs;
}
```

Figure 6: Implementation of the MapReduce programming model in Skywriting. The user provides a list of inputs, a mapper function, a reducer function and the number of reducers to use.

The `mapreduce()` function first applies `mapper` to each element of `inputs`. `mapper` is a Skywriting function that returns a list of `r` elements. The map outputs are then shuffled, so that the ith output of each map becomes an input to the ith reduce. Finally, the `reducer` function is applied `r` times to the collected reduce inputs. In typical use, the inputs to `mapreduce()` are data objects containing the input splits, and the `mapper` and `reducer` functions invoke `spawn_exec()` to perform computation in another language.

Note that the `mapper` function is responsible for partitioning data amongst the reducers, and the `reducer` function must merge the inputs that it receives. The implementation of `mapper` may also incorporate a combiner, if desired [18]. To simplify development, we have ported portions of the Hadoop MapReduce framework to run as CIEL tasks, and provide helper functions for partitioning, merging, and processing Hadoop file formats.

Any higher-level language that is compiled into a DAG of tasks can also be compiled into a Skywriting program, and executed on a CIEL cluster. For example, one could develop Skywriting back-ends for Pig [32] and DryadLINQ [39], raising the possibility of extending those languages with support for unbounded iteration.

5 Implementation issues

The current implementation of CIEL and Skywriting contains approximately 9,500 lines of Python code, and a few hundred lines of C, Java and other languages in the executor bindings. All of the source code, along with a suite of example Skywriting programs (including those used to evaluate the system in §6), is available to download from our project website:
`http://www.cl.cam.ac.uk/netos/ciel/`

The remainder of this section describes three interesting features of our implementation: memoisation (§5.1), master fault tolerance (§5.2) and streaming (§5.3).

5.1 Deterministic naming & memoisation

Recall that all objects in a CIEL cluster have a *unique* name. In this subsection, we show how an appropriate choice of names can enable memoisation.

Our original implementation of CIEL used globally-unique identifiers (UUIDs) to identify all data objects. While this was a conceptually simple scheme, it complicated fault tolerance (see following subsection), because the master had to record the generated UUIDs to support deterministic task replay after a failure.

This motivated us to reconsider the choice of names. To support fault-tolerance, existing systems assume that individual tasks are deterministic [18, 26], and CIEL makes the same assumption (§3.1). It follows that two tasks with the same dependencies—including the executable code as a dependency—will have identical behaviour. Therefore the n outputs of a task created with the following Skywriting statement

```
result = spawn_exec(executor, args, n);
```

will be completely determined by `executor`, `args`, `n` and their indices. We could therefore construct a name for the ith output by concatenating `executor`, `args`, `n` and i, with appropriate delimiters. However, since `args` may itself contain references, names could grow to an unmanageable length. We therefore use a collision-resistant hash function, $\mathcal{H}$, to compute a digest of `args` and `n`, which gives the resulting name:

executor	:	$\mathcal{H}$(args‖n)	:	i

We currently use the 160-bit SHA-1 hash function to generate the digest.

Recall the lazy evaluation algorithm from §3.2: tasks are only executed when their expected outputs are needed to resolve a dependency for a blocked task. If a new task's outputs have already been produced by a previous task, the new task need not be executed at all. Hence, as a result of deterministic naming, CIEL memoises task results, which can improve the performance of jobs that perform repetitive tasks.

The goals of our memoisation scheme are similar to the recent Nectar system [23]. Nectar performs static

analysis on DryadLINQ queries to identify subqueries that have previously been computed on the same data. Nectar is implemented at the DryadLINQ level, which enables it to make assumptions about the semantics of the each task, and the cost/benefit ratio of caching intermediate results. For example, Nectar can re-use the results of commutative and associative aggregations from a previous query, if the previous query operated on a prefix of the current query's input. The expressiveness of CIEL jobs makes it more challenging to run these analyses, and we are investigating how simple annotations in a Skywriting program could provide similar functionality in our system.

5.2 Fault tolerance

A distributed execution engine must continue to make progress in the face of network and computer faults. As jobs become longer—and, since CIEL allows unbounded iteration, they may become extremely long—the probability of experiencing a fault increases. Therefore, CIEL must tolerate the failure of any machine involved in the computation: the client, workers and master.

Client fault tolerance is trivial, since CIEL natively supports iterative jobs and manages job execution from start to finish. The client's only role is to submit the job: if the client subsequently fails, the job will continue without interruption. By contrast, in order to execute an iterative job using a non-iterative framework, the client must run a driver program that performs all data-dependent control flow (such as convergence testing). Since the driver program executes outside the framework, it does not benefit from transparent fault tolerance, and the developer must provide this manually, for example by checkpointing the execution state. In our system, a Skywriting script replaces the driver program, and CIEL executes the whole script reliably.

Worker fault tolerance in CIEL is similar to Dryad [26]. The master receives periodic heartbeat messages from each worker, and considers a worker to have failed if (*i*) it has not sent a heartbeat after a specified timeout, and (*ii*) it does not respond to a reverse message from the master. At this point, if the worker has been assigned a task, that task is deemed to have failed.

When a task fails, CIEL automatically re-executes it. However, if it has failed because its inputs were stored on a failed worker, the task is no longer runnable. In that case, CIEL recursively re-executes predecessor tasks until all of the failed task's dependencies are resolved. To achieve this, the master invalidates the locations in the object table for each missing input, and lazily re-evaluates the missing inputs. Other tasks that depend on data from the failed worker will also fail, and these are similarly re-executed by the master.

Master fault tolerance is also supported in CIEL. In MapReduce and Dryad, a job fails completely if its master process fails [18, 26]; in Hadoop, all jobs fail if the JobTracker fails [2]; and master failure will usually cause driver programs that submit multiple jobs to fail. However, in CIEL, all master state can be derived from the set of active jobs. At a minimum, persistently storing the root task of each active job allows a new master to be created and resume execution immediately. CIEL provides three complementary mechanisms that extend master fault tolerance: *persistent logging*, *secondary masters* and *object table reconstruction*.

When a new job is created, the master creates a log file for the job, and synchronously writes its root task descriptor to the log. By default, it writes the log to a log directory on local secondary storage, but it can also write to a networked file system or distributed storage service. As new tasks are created, their descriptors are appended asynchronously to the log file, and periodically flushed to disk. When the job completes, a concrete reference to its result is written to the log directory. Upon restarting, the master scans its log directory for jobs without a matching result. For those jobs, it replays the log, rebuilding the dynamic task graph, and ignoring the final record if it is truncated. Once all logs have been processed, the master restarts the jobs by lazily evaluating their outputs.

Alternatively, the master may log state updates to a *secondary master*. After the secondary master registers with the primary master, the primary asynchronously forwards all task table and object table updates to the secondary. Each new job is sent synchronously, to ensure that it is logged at the secondary before the client receives an acknowledgement. In addition, the secondary records the address of every worker that registers with the primary, so that it can contact the workers in a fail-over scenario. The secondary periodically sends a heartbeat to the primary; when it detects that the primary has failed, the secondary instructs all workers to re-register with it. We evaluate this scenario in §6.5.

If the master fails and subsequently restarts, the workers can help to reconstruct the object table using the contents of their local object stores. A worker deems the master to have failed if it does not respond to requests. At this point, the worker switches into *reregister* mode, and the heartbeat messages are replaced with periodic registration requests to the same network location. When the worker finally contacts a new master, the master pulls a list of the worker's data objects, using a protocol based on GFS master recovery [22].

5.3 Streaming

Our earlier definition of a task (§3.1) stated that a task produces data objects as part of its *result*. This definition

implies that object production is atomic: an object either exists completely or not at all. However, since data objects may be very large, there is often the opportunity to *stream* the partially-written object between tasks, which can lead to pipelined parallelism.

If the producing task has streamable outputs, it sends a *pre-publish* message to the master, containing *stream references* for each streamable output. These references are used to update the object table, and may unblock other tasks: the *stream consumers*. A stream consumer executes as before, but the executed code reads its input from a named pipe rather than a local file. A separate thread in the consuming worker process fetches chunks of input from the producing worker, and writes them into the pipe. When the producer terminates successfully, it commits its outputs, which signals to the consumer that no more data remains to be read.

In the present implementation, the stream producer also writes its output data to a local disk, so that, if the stream consumer fails, the producer is unaffected. If the producer fails while it has a consumer, the producer rolls back any partially-written output. In this case, the consumer will fail due to missing input, and trigger re-execution of the producer (§5.2). We are investigating more sophisticated fault-tolerance and scheduling policies that would allow the producer and consumer to communicate via direct TCP streams, as in Dryad [26] and the Hadoop Online Prototype [16]. However, as we show in the following section, support for streaming yields useful performance benefits for some applications.

6 Evaluation

Our main goal in developing CIEL was to develop a system that supports a more powerful model of computation than existing distributed execution engines, without incurring a high cost in terms of performance. In this section, we evaluate the performance of CIEL running a variety of applications implemented in Skywriting. We investigate the following questions:

1. How does CIEL's performance compare to a system in production use (viz. Hadoop)? (§6.1, §6.2)
2. What benefits does CIEL provide when executing an iterative algorithm? (§6.2)
3. What overheads does CIEL impose on compute-intensive tasks? (§6.3, §6.4)
4. What effect does master failure have on end-to-end job performance? (§6.5)

For our evaluation, we selected a set of algorithms to answer these questions, including MapReduce-style, iterative, and compute-intensive algorithms. We chose dynamic programming algorithms to demonstrate CIEL's ability to execute algorithms with data dependencies that do not translate to the MapReduce model.

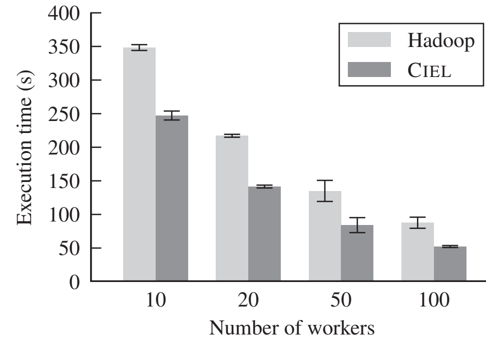


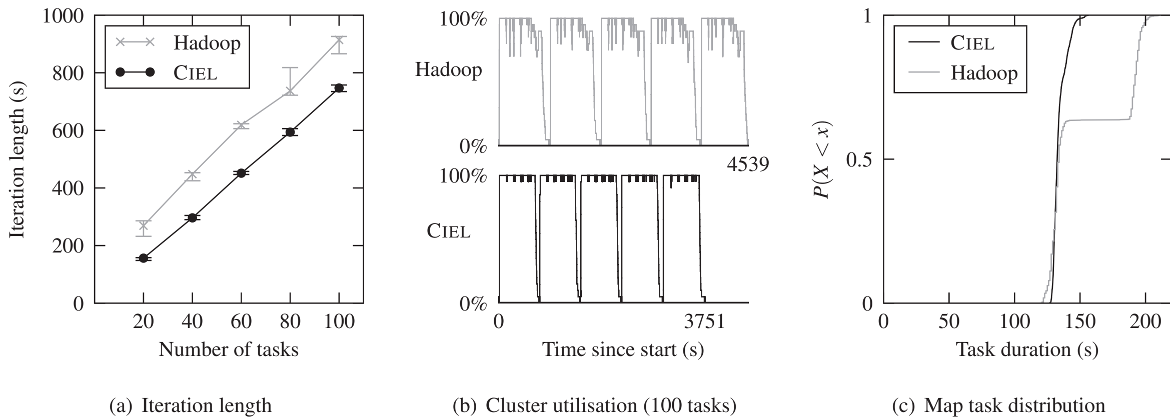
Figure 7: Grep execution time on Hadoop and CIEL (§6.1).

All of the results presented in this section were gathered using `m1.small` virtual machines on the Amazon EC2 cloud computing platform. At the time of writing, an `m1.small` instance has 1.7 GB of RAM and 1 virtual core (equivalent to a 2007 AMD Opteron or Intel Xeon processor) [1]. In all cases, the operating system was Ubuntu 10.04, using Linux kernel version 2.6.32 in 32-bit mode. Since the virtual machines are single-core, we run one CIEL worker per machine, and configure Hadoop to use one map slot per TaskTracker.

6.1 Grep

Our grep benchmark uses the Grep example application from Hadoop to search a 22.1 GB dump of English-language Wikipedia for a three-character string. The original Grep application performs two MapReduce jobs: the first job parses the input data and emits the matching strings, and the second sorts the matching strings by frequency. In Skywriting, we implemented this as a single script that uses two invocations of `mapreduce()` (§4.3). Both systems use identical data formats and execute an identical computation (regular expression matching).

Figure 7 shows the absolute execution time for Grep as the number of workers increases from 10 to 100. Averaged across all runs, CIEL outperforms Hadoop by 35%. We attribute this to the Hadoop heartbeat protocol, which limits the rate at which TaskTrackers poll for tasks once every 5 seconds, and the mandatory "setup" and "cleanup" phases that run at the start and end of each job [38]. As a result, the relative performance of CIEL improves as the job becomes shorter: CIEL takes 29% less time on 10 workers, and 40% less time on 100



(a) Iteration length (b) Cluster utilisation (100 tasks) (c) Map task distribution

Figure 8: Results of the k-means experiment on Hadoop and CIEL with 20 workers (§6.2).

workers. We observed that a no-op Hadoop job (which dispatches one map task per worker, and terminates immediately) runs for an average of 30 seconds. Since Grep involves two jobs, we would not expect Hadoop to complete the benchmark in less than 60 seconds. These results confirm that Hadoop is not well-suited to short jobs, which is a result of its original application (large-scale document indexing). However, anecdotal evidence suggests that production Hadoop clusters mostly run jobs lasting less than 90 seconds [40].

6.2 k-means

We ported the Hadoop-based k-means implementation from the Apache Mahout scalable machine learning toolkit [3] to CIEL. Mahout simulates iterative-algorithm support on Hadoop by submitting a series of jobs and performing a convergence test outside the cluster; our port uses a Skywriting script that performs all iterations and convergence testing in a single CIEL job.

In this experiment, we compare the performance of the two versions by running 5 iterations of clustering on 20 workers. Each task takes 64 MB of input—80,000 dense vectors, each containing 100 double-precision values—and $k = 100$ cluster centres. We increase the number of tasks from 20 to 100, in multiples of the cluster size. As before, both systems use identical data formats and execute an identical computational kernel. Figure 8(a) compares the per-iteration execution time for the two versions. For each job size, CIEL is faster than Hadoop, and the difference ranges between 113 and 168 seconds. To investigate this difference further, we now analyse the task execution profile.

Figure 8(b) shows the cluster utilisation as a function of time for the 5 iterations of 100 tasks. From this figure, we can compute the average cluster utilisation: i.e. the probability that a worker is assigned a task at any point during the job execution. Across all job sizes, CIEL achieves 89 $\pm$ 2% average utilisation, whereas Hadoop achieves 84% utilisation for 100 tasks (and only 59% utilisation for 20 tasks). The Hadoop utilisation drops to 70% at several points when there is still runnable work, which is visible as troughs or "noise" in the utilisation time series. This scheduling delay is due to Hadoop's polling-based implementation of task dispatch.

CIEL also achieves higher utilisation in this experiment because the task duration is less variable. The execution time of k-means is dominated by the map phase, which computes k Euclidean distances for each data point. Figure 8(c) shows the cumulative distribution of map task durations, across all k-means experiments. The Hadoop distribution is clearly bimodal, with 64% of the tasks being "fast" (μ = 130.9, σ = 3.92) and 36% of the tasks being "slow" (μ = 193.5, σ = 3.71). By contrast, all of the CIEL tasks are "fast" (μ = 134.1, σ = 5.05). On closer inspection, the slow Hadoop tasks are non-data-local: i.e. they read their input from another HDFS data node. When computing an iterative job such as k-means, CIEL can use information about previous iterations to improve the performance of subsequent iterations. For example, CIEL preferentially schedules tasks on workers that consumed the same inputs in previous iterations, in order to exploit data that might still be stored in the page cache. When a task reads its input from a remote worker, CIEL also updates the object table to record that another replica of that input now exists. By contrast, each iteration on Hadoop is an independent job, and Hadoop does not perform cross-job optimisations, so the scheduler is less able to exploit data locality.

In the CIEL version, a Skywriting task performs a convergence test and, if necessary, spawns a subsequent iteration of k-means. However, compared to the data-intensive map phase, its execution time is insignificant: in the 100-task experiment, less than 2% of the total job

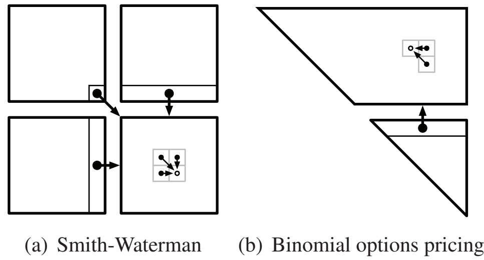

(a) Smith-Waterman (b) Binomial options pricing

Figure 9: Smith-Waterman (§6.3) and BOPM (§6.4) are dynamic programming algorithms, with macro-level (partition) and micro-level (element) dependencies.

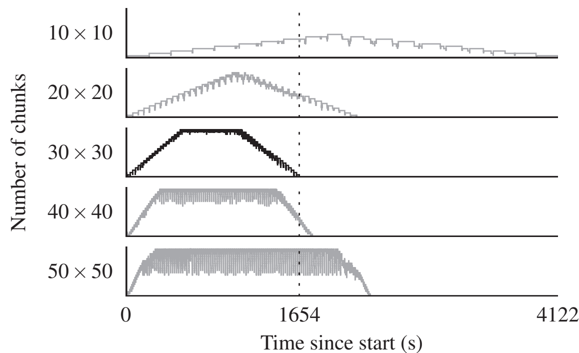


Figure 10: Smith-Waterman cluster utilisation against time, for different block granularities. The best performance is observed with 30 × 30 blocks.

execution time is spent running Skywriting tasks. The Skywriting execution time is dominated by communication with the master, as the script sends a new task descriptor to the master for each task in the new iteration.

6.3 Smith-Waterman

In this experiment, we evaluate strategies for parallelising the Smith-Waterman sequence alignment algorithm [36]. For strings of size m and n, the algorithm computes mn elements of a dynamic programming matrix. However, since each element depends on three predecessors, the algorithm is not embarrassingly parallel. We divide the matrix into blocks—where each block depends on values from its three neighbours (Figure 9(a))—and process one block per task.

We use CIEL to compute the alignment between two 1 MB strings on 20 workers. Figure 10 shows the cluster utilisation as the block granularity is varied: a granularity of $m \times n$ means that the computation is split into mn blocks. For 10 × 10 (the most coarse-grained case that we consider), the maximum degree of parallelism is 10, because the dependency structure limits the maximum achievable parallelism to the length of the antidiagonal in the block matrix. Increasing the number of blocks to 20 × 20 allows CIEL to achieve full utilisation briefly, but performance remains poor because the majority of the job duration is spent either ramping up to or down from full utilisation. We observe the best performance for 30 × 30, which ramps up to full utilisation more quickly than coarser-grained configurations, and maintains full utilisation for an extended period, because there are more runnable tasks than workers. Increasing the granularity beyond 30 × 30 leads to poorer overall performance, because the overhead of task dispatch becomes a significant fraction of task duration. Furthermore, the scheduler cannot dispatch tasks quickly enough to maintain full utilisation, which appears as "noise" in Figure 10.

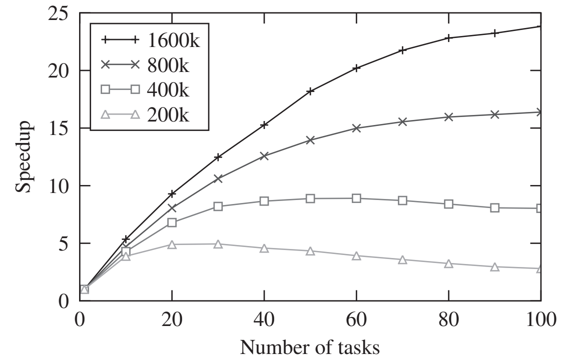


Figure 11: Speedup of BOPM (§6.4) on 47 workers as the number of tasks is varied and the resolution is increased.

6.4 Binomial options pricing

We now consider another dynamic programming algorithm: the binomial options pricing model (BOPM) [17]. BOPM computes a binomial tree, which can be represented as an upper-triangular matrix, P. The rightmost column of P can be computed directly from the input parameters, after which element $p_{i,j}$ depends on $p_{i,j+1}$ and $p_{i+1,j+1}$, and the result is the value of $p_{1,1}$. We achieve parallelism by dividing the matrix into row chunks, creating one task per chunk, and *streaming* the top row of each chunk into the next task. Figure 9(b) shows the element- and chunk-level data dependencies for this algorithm.

BOPM is not an embarrassingly parallel algorithm. However, we expect CIEL to achieve some speedup, since rows of the matrix can be computed in parallel, and we can use streaming tasks (§5.3) to obtain pipelined parallelism. We can also achieve better speedup by increasing the resolution of the calculation: the problem size

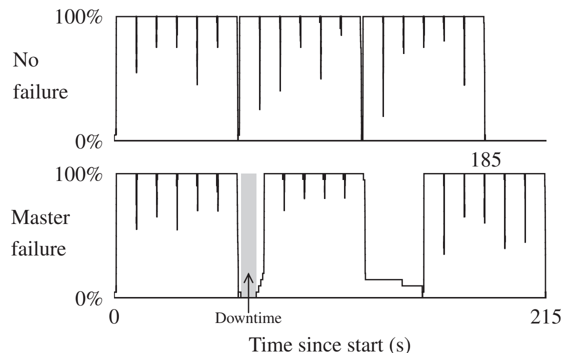


Figure 12: Cluster utilisation for three iterations of an iterative algorithm (§6.5). In the lower case, the primary master fails over to a secondary at the beginning of the second iteration. The total downtime is 7.7 seconds.

(n) is inversely proportional to the time step (Δt), and the serial execution time increases as $O(n^2)$.

Figure 11 shows the parallel speedup of BOPM on a 47-worker CIEL cluster. We vary the number of tasks, and increase n from 2×10^5 to 1.6×10^6. As expected, the maximum speedup increases as the problem size grows, because the amount of independent work in each task grows. For $n = 2 \times 10^5$ the maximum speedup observed is 4.9×, whereas for $n = 1.6 \times 10^6$ the maximum speedup observed is 23.8×. After reaching the maximum, the speedup decreases as more tasks are added, because small tasks suffer proportionately more from constant per-task overhead. Due to our streaming implementation, the minimum execution time for a stream consumer is approximately one second. We plan to replace our simple, polling-based streaming implementation with direct TCP sockets, which will decrease the per-task overhead and improve the maximum speedup.

6.5 Fault tolerance

Finally, we conducted an experiment in which master fail-over was induced during an iterative computation. Figure 12 contrasts the cluster utilisation in the non-failure and master-failure cases, where the master fail-over occurs at the beginning of the second iteration. Between the failure of the primary master and the resumption of execution, 7.7 seconds elapse: during this time, the secondary master must detect primary failure, contact all of the workers, and wait until the workers register with the secondary. Utilisation during the second iteration is poorer, because some tasks must be replayed due to the failure. The overall job execution time increases by 30 seconds, and the original full utilisation is attained once more in the third iteration.

7 Alternative approaches

CIEL was inspired primarily by the MapReduce and Dryad distributed execution engines. However, there are several different and complementary approaches to large-scale distributed computing. In this section, we briefly survey the related work from different fields.

7.1 High performance computing (HPC)

The HPC community has long experience in developing parallel programs. OpenMP is an API for developing parallel programs on shared-memory machines, which has recently added support for task parallelism with dependencies [7]. In this model, a task is a C or Fortran function marked with a compiler directive that identifies the formal parameters as task inputs and outputs. The inputs and outputs are typically large arrays that fit completely in shared memory. OpenMP is more suitable than CIEL for jobs that share large amounts of data that is frequently updated on a fine-grained basis. However, the parallel efficiency of a shared memory system is limited by interconnect contention and/or non-uniform memory access, which limits the practical size of an OpenMP job. Nevertheless, we could potentially use OpenMP to exploit parallelism within an individual multi-core worker.

Larger HPC programs typically use the Message Passing Interface (MPI) for parallel computing on distributed memory machines. MPI provides low-level primitives for sending and receiving messages, collective communication and synchronisation [30]. MPI is optimised for low-latency supercomputer interconnects, which often have a three-dimensional torus topology [35]. These interconnects are optimal for problems that decompose spatially and have local interactions with neighbouring processors. Since these interconnects are highly reliable, MPI does not tolerate intermittent message loss, and so checkpointing is usually used for fault tolerance. For example, Piccolo, which uses MPI, must restart an entire computation from a checkpoint if an error occurs [34].

7.2 Programming languages

Various programming paradigms have been proposed to simplify or fully automate software parallelisation.

Several projects have added parallel language constructs to existing programming languages. Cilk-NOW is a distributed version of Cilk that allows developers to `spawn` a C function on another cluster machine and `sync` on its result [11]. X10 is influenced by Java, and provides `finish` and `async` blocks that allow developers to implement more general synchronisation patterns [15]. Both implement *strict multithreading*, which restricts synchronisation to between a spawned thread

and its ancestor [10]. While this does not limit the expressiveness of these languages, it necessitates additional synchronisation in the implementation of, for example, MapReduce, where non-ancestor tasks may synchronise.

Functional programming languages offer the prospect of fully automatic parallelism [8]. NESL contains a parallel "apply to each" operator (i.e. a `map()` function) that processes the elements of a sequence in parallel, and the implementation allows nested invocation of this operator [9]. Glasgow Distributed Haskell contains mechanisms for remotely evaluating an expression on a particular host [33]. Though theoretically appealing, parallel functional languages have not demonstrated as great scalability as MapReduce or Dryad, which sacrifice expressivity for efficiency.

7.3 Declarative programming

The relational algebra, which comprises a relatively small set of operators, can be parallelised in time (pipelining) and space (partitioning) [19]. Pig and Hive implement the relational algebra using a DAG of MapReduce jobs on Hadoop [32, 37]; DryadLINQ and SCOPE implement it using a Dryad graph [14, 39].

The relational algebra is not universal but can be made more expressive by adding a least fixed point operator [5], and this research culminated in support for recursive queries in SQL:1999 [20]. Recently, Bu *et al.* showed how some recursive SQL queries may be translated to iterative Hadoop jobs [12].

Datalog is a declarative query language based on first-order logic [13]. Recently, Alvaro *et al.* developed a version of Hadoop and the Hadoop Distributed File System using Overlog (a dialect of Datalog), and demonstrated that it was almost as efficient as the equivalent Java code, while using far fewer lines of code [6]. We are not aware of any project that has used a fully-recursive logic-programming language to implement data-intensive programs, though the non-recursive Cascalog language, which runs on Hadoop, is a step in this direction [29].

7.4 Distributed operating systems

Hindman *et al.* have developed the Mesos distributed operating system to support "diverse cluster computing frameworks" on a shared cluster [25]. Mesos performs fine-grained scheduling and fair sharing of cluster resources between the frameworks. It is predicated on the idea that no single framework is suitable for all applications, and hence the resources must be virtualised to support different frameworks at once. By contrast, we have designed CIEL with primitives that support any form of computation (though not always optimally), and allow frameworks to be virtualised at the language level.

8 Conclusions

We designed CIEL to provide a superset of the features that existing distributed execution engines provide. With Skywriting, it it possible to write iterative algorithms in an imperative style and execute them with transparent fault tolerance and automatic distribution. However, CIEL can also execute any MapReduce job or Dryad graph, and the support for iteration allows it to perform Pregel- and Piccolo-style computations.

Our next step is to integrate CIEL primitives with existing programming languages. At present, only Skywriting scripts can create new tasks. This does not limit universality, but it requires developers to rewrite their driver programs in Skywriting. It can also put pressure on the Skywriting runtime, because all scheduling-related control-flow decisions must ultimately pass through interpreted code. The main benefit of Skywriting is that it masks the complexity of continuation-passing style behind the dereference operator (§4.2). We now seek a way to extend this abstraction to mainstream programming languages.

CIEL scales across hundreds of commodity machines, but other scaling challenges remain. For example, it is unclear how best to exploit multiple cores in a single machine, and we currently pass this problem to the executors, which receive full use of an individual machine. This gives application developers fine control over how their programs execute, at the cost of additional complexity. However, it limits efficiency if tasks are inherently sequential and multiple cores are available. Furthermore, the I/O saving from colocating a stream producer and its consumers on a single host may outweigh the cost of CPU contention. Finding the optimal schedule is a hard problem, and we are investigating simple annotation schemes and heuristics that improve performance in the common case. The recent work on cluster operating systems and scheduling algorithms [25, 27] offers hope that this problem will admit an elegant solution.

Further information about CIEL and Skywriting, including the source code, a language reference and a tutorial, is available from the project website:
`http://www.cl.cam.ac.uk/netos/ciel/`

Acknowledgements

We wish to thank our past and present colleagues in the Systems Research Group at the University of Cambridge for many fruitful discussions that contributed to the evolution of CIEL. We would also like to thank Byung-Gon Chun, our shepherd, and the anonymous reviewers, whose comments and suggestions have been invaluable for improving the presentation of this work.

References

[1] Amazon EC2. `http://aws.amazon.com/ec2/`.

[2] Apache Hadoop. `http://hadoop.apache.org/`.

[3] Apache Mahout. `http://mahout.apache.org/`.

[4] JSON. `http://www.json.org/`.

[5] AHO, A. V., AND ULLMAN, J. D. Universality of data retrieval languages. In *Proceedings of POPL* (1979).

[6] ALVARO, P., CONDIE, T., CONWAY, N., ELMELEEGY, K., HELLERSTEIN, J. M., AND SEARS, R. BOOM Analytics: Exploring data-centric, declarative programming for the cloud. In *Proceedings of EuroSys* (2010).

[7] AYGUADÉ, E., COPTY, N., DURAN, A., HOEFLINGER, J., LIN, Y., MASSAIOLI, F., TERUEL, X., UNNIKRISHNAN, P., AND ZHANG, G. The design of OpenMP tasks. *IEEE Trans. Parallel Distrib. Syst. 20*, 3 (2009), 404–418.

[8] BACKUS, J. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM 21*, 8 (1978), 613–641.

[9] BLELLOCH, G. E. Programming parallel algorithms. *Commun. ACM 39*, 3 (1996), 85–97.

[10] BLUMOFE, R. D., AND LEISERSON, C. E. Scheduling multithreaded computations by work stealing. *J. ACM 46*, 5 (1999), 720–748.

[11] BLUMOFE, R. D., AND LISIECKI, P. A. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of USENIX ATC* (1997).

[12] BU, Y., HOWE, B., BALAZINSKA, M., AND ERNST, M. D. HaLoop: Efficient iterative data processing on large clusters. In *Proceedings of VLDB* (2010).

[13] CERI, S., GOTTLOB, G., AND TANCA, L. What you always wanted to know about Datalog (and never dared to ask). *IEEE Trans. on Knowl. and Data Eng. 1*, 1 (1989), 146–166.

[14] CHAIKEN, R., JENKINS, B., LARSON, P.-A., RAMSEY, B., SHAKIB, D., WEAVER, S., AND ZHOU, J. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of VLDB* (2008).

[15] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of OOPSLA* (2005).

[16] CONDIE, T., CONWAY, N., ALVARO, P., HELLERSTEIN, J. M., ELMELEEGY, K., AND SEARS, R. MapReduce Online. In *Proceedings of NSDI* (2010).

[17] COX, J. C., ROSS, S. A., AND RUBINSTEIN, M. Option pricing: A simplified approach. *Journal of Financial Economics 7*, 3 (1979), 229–263.

[18] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI* (2004).

[19] DEWITT, D., AND GRAY, J. Parallel database systems: The future of high performance database systems. *Commun. ACM 35*, 6 (1992), 85–98.

[20] EISENBERG, A., AND MELTON, J. SQL: 1999, formerly known as SQL3. *SIGMOD Rec. 28*, 1 (1999), 131–138.

[21] EKANAYAKE, J., PALLICKARA, S., AND FOX, G. MapReduce for data intensive scientific analyses. In *Proceedings of eScience* (2008).

[22] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of SOSP* (2003).

[23] GUNDA, P. K., RAVINDRANATH, L., THEKKATH, C. A., YU, Y., AND ZHUANG, L. Nectar: Automatic management of data and computation in data centers. In *Proceedings of OSDI* (2010).

[24] HALSTEAD, JR., R. H. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[25] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI* (2011).

[26] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys* (2007).

[27] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of SOSP* (2009).

[28] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of SIGMOD* (2010).

[29] MARZ, N. Introducing Cascalog. `http://nathanmarz.com/blog/introducing-cascalog/`.

[30] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard. Tech. Rep. CS-94-230, University of Tennessee, 1994.

[31] MURRAY, D. G., AND HAND, S. Scripting the cloud with Skywriting. In *Proceedings of HotCloud* (2010).

[32] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of SIGMOD* (2008).

[33] POINTON, R. F., TRINDER, P. W., AND LOIDL, H.-W. The design and implementation of Glasgow Distributed Haskell. In *Proceedings of IFL* (2001).

[34] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of OSDI* (2010).

[35] SCOTT, S. L., AND THORSON, G. M. The Cray T3E network: adaptive routing in a high performance 3D torus. In *Proceedings of HOT Interconnects* (1996).

[36] SMITH, T., AND WATERMAN, M. Identification of common molecular subsequences. *Journal of molecular biology 147*, 1 (1981), 195–197.

[37] THUSOO, A., SARMA, J. S., JAIN, N., SHAO, Z., CHAKKA, P., ZHANG, N., ANTONY, S., LIU, H., AND MURTHY, R. Hive: A petabyte scale data warehouse using Hadoop. In *Proceedings of ICDE* (2010).

[38] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[39] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ÚLFAR ERLINGSSON, GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of OSDI* (2008).

[40] ZAHARIA, M., BORTHAKUR, D., SEN SARMA, J., ELMELEEGY, K., SHENKER, S., AND STOICA, I. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys* (2010).

A Semantic Framework for Data Analysis in Networked Systems *

Arun Viswanathan[†] Alefiya Hussain[†‡] Jelena Mirkovic[†] Stephen Schwab[‡] John Wroclawski[†]

[†] *USC/Information Sciences Institute* [‡] *Sparta Inc.*

{*aviswana, hussain, mirkovic, jtw*}*@isi.edu* *Stephen.Schwab@cobham.com*

Abstract

Effective analysis of raw data from networked systems requires bridging the semantic gap between the data and the user's high-level understanding of the system. The raw data represents facts about the system state and analysis involves identifying a set of semantically relevant *behaviors*, which represent "interesting" relationships between these facts. Current analysis tools, such as wireshark and splunk, restrict analysis to the low-level of individual facts and provide limited constructs to aid users in bridging the semantic gap. Our objective is to enable semantic analysis at a level closer to the user's understanding of the system or process. The key to our approach is the introduction of a logic-based formulation of high-level *behavior* abstractions as a sequence or a group of related facts. This allows treating behavior representations as fundamental analysis primitives, elevating analysis to a higher semantic-level of abstraction. In this paper, we propose a behavior-based semantic analysis framework which provides: (a) a formal language for modeling high-level assertions over networked systems data as *behavior models*, (b) an analysis engine for extracting instances of user-specified behavior models from raw data. Our approach emphasizes reuse, composibility and extensibility of abstractions. We demonstrate the effectiveness of our approach by applying it to five analyses tasks; modeling a hypothesis on traffic traces, modeling experiment behavior, modeling a security threat, modeling dynamic change and composing higher-level models. Finally, we discuss the performance of our framework in terms of behavior complexity and number of input records.

*This work is funded by the Department of Homeland Security and Space and Naval Warfare Systems Center, San Diego, under Contract No. N66001-10-C-2018. All findings and conclusions expressed in this material are those of the authors and do not reflect the views of the funding agencies.
Part of Alefiya Hussain's contributions to this paper were while she was at Sparta Inc.

1 Introduction

The ability to convert raw data into higher-level insights and understanding has become a key enabler in many fields. We approach one particular aspect of this problem, namely the analysis of data within the domain of networked and distributed systems. Such systems routinely generate a plethora of logs, trace and audit data during their operation. Users, such as researchers and system administrators, use this raw data to understand system behavior, diagnose problems, discover new behaviors, or verify hypotheses. Effective analysis of such raw data requires bridging the semantic gap between raw data and the user's high-level understanding of the analysis domain. Our experience with analysis tools reveals that this problem is ill-addressed.

A typical approach to data analysis involves the user sifting through the data using simple search and correlation constructs like boolean queries to identify relationships and infer meaning from data. For example, wireshark [19] can help identify complete or incomplete TCP flows from packet traces and splunk [16] can help identify spurious logins from a server log. Our study of four popular tools, discussed in Section 2.1, reveals that current approaches require cumbersome multi-step analyses to infer semantic relationships from data. For example, a user analyzing a network packet trace may first have to extract individual flows by specifying specific attribute values related to each flow, and then somehow manually infer relationships like concurrency between the flows. This problem is further complicated if the user has to reason and analyze over multiple types of data. This separation between the raw data and the meaning it carries constitutes the semantic gap.

In this paper, we focus on the problem of expressing analyses tasks that are meaningful and useful to the user. Specifically, given a finite, timestamped list of *facts* about the system under observation, our objective is to assist the user in expressing and modeling semantically

relevant *behaviors*, which are “interesting” relationships between these facts or sequence of facts. These relationships encompass notions of ordering, causality, dependence, or concurrency.

Our insight is that higher-level understanding in networked and distributed systems can be expressed in the form of relationships between system states, simple behaviors, and complex behaviors. For example, in most situations, a typical web-server operation is better understood as a concurrent relationship between multiple HTTP sessions to a server rather than the details of the protocols and specific values in the packet headers. Thus, our data analysis approach introduces a *behavior* as a primitive analysis construct. Behaviors can be extended or constrained to create a *behavior model*, which forms an assertion about the overall behavior of the system. A behavior model can then be rapidly applied over data to validate the assertion. We discuss complete details about specifying behavior models in Section 3, and Section 4 presents the analysis engine for extracting instances of user-specified behavior models from raw data.

The behavior models are abstract entities to capture the semantic essence of a particular relationship without focusing on unnecessary details or particular parameters that may vary between individual facts or behaviors. Incorporation of abstract behavior models as explicitly represented and manipulated constructs within our framework provides two key benefits. First, this abstraction allows users of our framework to analyze and understand the raw data at a semantically relevant level. In Section 3.4, we introduce an example of a behavior model to identify pairs of communication events where the destination IP of the second event is same as the source IP of the first. Such models can be used to analyze many different datasets without any modification. Additionally, since behavior models are primitive analysis constructs, the framework supports extensibility by composing new models from behavior models present in the knowledge base as demonstrated in Section 5.5. Thus, representing analysis expertise explicitly as behavior models formalizes the semantics for data analysis in networked systems.

The second key benefit of our work is the ability to foster sharing and reuse of knowledge embedded in explicitly represented behavior models. Our first-hand experience with existing tools suggests that in most cases knowledge inferred from analysis resides either in a domain-specific tool or a single expert’s brain. This is due to a lack of an explicit representation for capturing, storing, sharing, and reusing such knowledge in a context-independent way. Many current tools are either static in nature, handling only a fixed set of analyses and record types, or may offer limited extensibility, but through some mechanism that involves significant effort. For example, wireshark [19] is easily extensible using plugins, but writing a plugin requires understanding the wireshark API and C programming skills. In contrast, a well defined shareable format for representing knowledge about networked systems data offers the prospect that many different tools can be driven by, and contribute to, a single shared knowledge base.

Beyond the basic challenge, the task of semantic-level analysis is difficult for two disparate reasons. First, the definition of “interesting” may vary widely in different situations, requiring a rich toolbox of techniques for effective analysis. We address this problem by restricting the definition of “interesting relationships” to expressing a particular set of characteristics of networked systems as discussed in Section 3.1. Second, in large scale systems, efficient and intelligent data analysis is extremely resource intensive due to the sheer volume of system events and traces. While in Section 6 we report performance results, this paper primarily discusses the fundamental aspects of defining and employing explicit behavior models as a data analysis tool. Real-time analysis of data for applications such as intrusion detection is a future goal as discussed in Section 7.

The fundamental contribution of this paper is the introduction of a behavior-based semantic analysis framework for *confirmatory* and *exploratory* analysis of multivariate, multi-type, timestamped data captured from networked systems. The main elements of the semantic framework include (a) a specialized formal language for specifying behavior models and (b) an analysis engine for extracting instances of user-specified behavior models from data. In confirmatory analysis, the user specifies a validation criteria, expected system behavior or hypothesis, by writing a specific model or through composing a high-level model from existing models contained within the knowledge base of the framework. In exploratory analysis, a user applies existing models from the knowledge base to explore data for new or unanticipated behaviors. In Section 5 we present five detailed examples of how the framework can be applied for these data analysis tasks.

2 Related Work

In this section, we set the context for our work by first studying four popular analysis tools followed by a discussion on specification-based approaches for analysis of networked systems data.

2.1 Tool Comparison

In this section, we study four popular analysis methodologies: wireshark v1.2.7 [19], splunk v4.1 [16], Simple Event Correlator (SEC) v2.5.3 [18], Bro v1.5.2 [14], and compare them with our behavior-based semantic analysis framework (SAF). Both wireshark and splunk are

	wireshark	splunk	SEC	Bro	SAF
System goals	Interactive analysis	Interactive analysis	Real-time event correlation	High-speed, real-time monitoring	Interactive analysis
Input data	Network packets	Ascii data from any source	Ascii data from files, stdin, pipes	Network packets	Any type of data (with plugin)
Specification language	Boolean logic	Boolean logic	Simple language for specifying rules	Bro scripting language	Formal language based on temporal logic, interval temporal logic and boolean logic
Primitive constructs	Boolean predicates	Boolean predicates, unix-like pipelines and commands	Boolean predicates, functions written in Perl	Events (low-level or higher-level)	Behavior (low-level or higher-level)
Semantic constructs	None	External commands can encode semantics	Perl functions can encode semantics	Network notions such as connections, IP addrs., ports, and network protocols	Temporal logic and interval temporal logic operators for defining behaviors (Section 3)
Composibility of specs	None	Queries can be recorded and then composed into other queries	Matching events can trigger creation of new high-level events	Policies can compose lower-level events to generate higher-level events	Behaviors can be composed into higher level behaviors
Abstraction	None	None	Limited	Yes	Yes

Table 1: Comparison of the behavior-based Semantic Analysis Framework (SAF) with four popular data analysis tools.

mainly interactive analysis tools while Bro and SEC are real-time monitoring tools. The behavior-based semantic analysis framework (SAF) falls in the category of interactive analysis tools. The tools are compared along seven dimensions in Table 1; (a) high-level goals, (b) input data types, (c) analysis specification language (d) primitive analysis constructs, (e) semantic analysis constructs, (f) ability to compose specifications and (g) abstraction, that is, specifications in terms of relationships between data attributes.

Each paragraph below introduces an analysis framework and the reader is directed to Table 1 for details. The corresponding features for our framework (SAF) are introduced in Table 1 and explored in future sections. We have not considered SQL-based approaches on streaming data for comparison [6], since SAF representations are at a higher-level of abstraction than database query languages. However, we further discuss how our framework could benefit by using the above SQL extensions to optimize event storage and retrieval in Section 7.

wireshark [19] is an open-source tool for interactive analysis of a large variety of network data from a packet capture file. Wireshark's design can be separated into the analysis framework and plugins. The analysis framework provides the ability to sift through large volumes of packets visually and provides a boolean query grammar for finding "interesting" relationships and statistical summaries over typical networking concepts, for example, rate, flows, bytes, and connections. The plugin architecture, on the other hand, is responsible for normalizing and presenting different types of packet data and protocol behavior to the analysis framework in a uniform way.

splunk [16] is a popular commercial framework for unified data analysis of a large variety of data. Splunk's strength comes from its ability to index various types of data, allowing the user to sift through logs by combining search queries using boolean operations, pipes and powerful statistical and aggregation functions. Splunk supports time-based, event-based, value-based correlations and also allows combining queries into higher-level queries. Splunk is extensible using apps, which allow encoding knowledge as queries for sharing and wider dissemination. However, it does not provide support for explicitly capturing domain expertise with semantic constructs. It does provide the ability to invoke external commands, thus providing an indirect way to incorporate explicit domain expertise into the analyses.

Simple Event Correlator(SEC) [18] is an open-source framework for rule-based event correlation. SEC reads the analysis specifications from a configuration file containing a set of event matching rules and corresponding actions. SEC processes data from log files, pipes and standard streams to trigger the configured actions on a match. It supports both time-based and event-based correlations and also allows specifying abstract rules that bind their values at runtime. SEC is more sophisticated than the previous two tools, it supports composing higher-level events by correlating low-level events, providing a framework for semantic understanding. Its rule-types *pair* and *pairwithwindow* capture some of the semantics of *ordering* and *duration*. However, it lacks support for inferring interval-based temporal relationships like concurrency and overlap and the analysis specification in the configuration files are not intuitive to capture

and share domain expertise in a generic way.

Bro [14] is a high-speed intrusion detection system for checking security policy violations by passively monitoring network traffic in real-time. Bro's security policies are written in the specialized Bro scripting language which is geared towards security analysis. The language supports semantic constructs such as connections, IP addresses, ports, and various network protocols along with various operators and functions to express different forms of network analyses. Bro has the ability to do time-based and event-based correlation. However, Bro mainly processes network packet data and uses a programming language-based analysis approach.

2.2 Specification-based Approaches

Specification-based approaches are particularly appealing in various areas of networked and distributed systems due to their ability to be abstract, concise, precise, and verifiable. In formal verification of distributed and concurrent systems, a system is specified in logic and then formal reasoning is applied on the specification to verify desired properties [3, 9]. In declarative networking, a specification language, Network Datalog (NDLog) [10], allows defining high-level networking specifications for rapidly specifying, modeling, implementing, and experimenting with evolving designs for network architectures. In testbed-based experimentation, a simple set of user-supplied expectations are used to validate expected behavior of an experiment [12].

The formal specification approaches have been well developed within the intrusion detection community and have been successfully applied to network and audit data for analysis. In this section we first present a brief overview of four such approaches and then compare them to SAF.

Roger et al. [15], leverage the idea that attack signatures are best expressed in simple temporal logic using temporal connectives to express ordering of events. They pose the detection problem as a model-checking problem against event logs. Naldurg et al. [13], propose another temporal-logic based approach for real-time monitoring and detection. Their language EAGLE supports parameterized recursive equations and allows specifying signatures with complex temporal event patterns along with properties involving real-time, statistics and data values. Kinder et al. [8], extend the logic CTL (Computation Tree Logic) and introduce CTPL (Computation Tree Predicate Logic) to describe malicious code as a high-level specification. Their approach allows writing specifications that capture malware variants. Ellis et al. [4], introduce a behavioral detection approach to malware by focusing on detecting patterns at higher-level of abstractions. They introduce three high-level behavioral signatures which have the ability to detect classes of worms without needing any apriori information of the worm behavior.

The SAF abstract models are comparable to the approaches of [13, 8, 4] in their use of formal logic and temporal constructs for specifications. But, in addition to providing an extended set of sophisticated intuitive operators and constructs, the behavior models presented in this paper can be generically applied to model various scenarios over a variety of data and are easily composed into semantically relevant higher-level models. This allows creating a knowledge base to explicitly capture domain expertise required for analyzing a large variety of operations encountered in networked and distributed systems as shown in Section 5. The higher-level behavioral signatures [4] based on the network-theoretic abstract communication network (ACN) are tightly bound to networking constructs like hosts, routers, sensors and links making them very restrictive in their ability to express general networked systems behaviors.

The SAF is based on a logic-based specification approach rather than a programming language-based specification approach like the one followed in Bro. Our goal is that the behavior models should be abstract but also concise and precise to support well-known knowledge representation and reasoning approaches. Logic is declarative and type-free, imparting formal semantics, abstract specifications, and efficient processing by analysis engines. The logic-based approach also enables building a knowledge base of behavior models to explicitly capture domain expertise that can be used to automatically reason and infer behavior models. However, logic-based approaches are less expressive than programming languages. The expressiveness of our approach is based on requirements derived from characteristics of networked systems as discussed in Section 3.1.

3 Behavior Models

A particular execution of a networked system or process can be captured as a sequence of *states*, where a state is a collection of attributes and their values. A *behavior* (b) is a sequence of one or more related states. A system execution is thus defined as a combination of different behaviors, and each new execution may generate a unique set of behaviors. A *behavior model* (ϕ) is a formula that makes an assertion about the overall behavior of the system.

For example, consider a simplified IP flow in networking, where a flow is a communication between two hosts identified by their IP addresses. For simplicity we assume an IP flow to be broken into two states: `ip_s2d` denotes a packet from some source to destination host and `ip_d2s` denotes a packet from a destination to source. Then, a valid IP flow behavior, `IPFLOW`, is one where `ip_s2d` and `ip_d2s` are related by their source

and destination attributes with the additional criteria that `ip_d2s` always occurs after `ip_s2d`. The behavior model (ϕ_{ipflow}) is an assertion that `IPFLOW` is valid. We discuss details of this example and extend it further in Section 3.4.

In this section, we first discuss the requirements and design choices for a language to specify behaviors followed by the formal syntax and semantics of the language.

3.1 Requirements

As discussed in Section 1, the key objective of our framework is to enable semantic-level analysis over data. A semantically expressive language for analysis over networked and distributed systems data must meet the following requirements: (a) enable analysis over multi-type, multi-variate, timestamped data, (b) express a wide variety of "interesting" relationships, (c) enable analysis over higher-level abstractions, and (d) enable composing abstractions into higher-level abstractions.

The language should express at-least the following "interesting" relationships to capture the core characteristics of networked and distributed systems: (a) causal relationships between behaviors, for example, a file being opened only if a user is authorized; (b) partial or total ordering, for example, in-order or out-of-order arrival of packets; (c) dynamic changes over time, for example, traffic between client and server drops after an attack on the server; (d) concurrency of operations, for example, simultaneous web client sessions; (e) multiple possible behaviors, for example, a polymorphic worm behavior may vary on each execution; (f) synchronous or asynchronous operations, for example, some operations need to complete within a specific time whereas others need not; (g) value dependencies between operations, for example, a TCP flow is valid only if the attribute–values contained in the individual packets are related to each other; (h) invariant operations, for example, some operations may always hold true and, (i) eventual operations, for example, some operations happen in the course of time. In addition, we need traditional mechanisms, such as boolean operators and loops, for combining these relationships into complex behaviors and mechanisms for basic counting of events and reasoning over the counts.

We do not claim completeness of the above requirements but we believe that being able to express the above classes of primitive relationships and combining them to form complex relationships would suffice for a wide range of situations, a few of which we demonstrate as case studies in Section 5.

3.2 Design

The following four design decisions realize the requirements listed above. First, our framework provides logic-based support to formulate behavior abstractions as a sequence or group of related events, where events are uniform representation of system facts as discussed later. This formulation allows treating this behavior representation as fundamental analysis primitive, elevating analyses to a higher semantic-level of abstraction.

Second, the language combines operators from Allen's interval-temporal logic [1], Lamport's Temporal Logic of Actions [9] and boolean logic. Temporal logic allows expressing the ordering of events in time without explicitly introducing time. Interval-temporal logic allows expressing relationships like concurrency, overlap and ordering between behaviors as relationships between their time-intervals. Additionally, complex behaviors are easily composed from simpler ones using boolean operators.

Third, the framework enables specifying dependency relationships between event attributes while leaving the values to be dynamically populated at runtime. Late binding enables abstract specifications that enrich the knowledge base as they can be directly applied to a wide variety of data-sets. This also enables parametrization of models during complex model composition as discussed in Section 5.5.

Lastly, the framework introduces the notion of a domain-independent *event* as a uniform representation of multi-type, multi-variate, timestamped data. Specifically, an *event* (e) is a representation of system state and is given by a 4-tuple $\langle o, c, t, av \rangle$ where o is the event-origin (for example, the host IP), c is the event-type (for example, PKT_TCP or APP_HTTPD), t is the event timestamp and $av = \{ \langle a_i, v_i \rangle \,|\, a_i \in A\,,\, v_i \in \mathsf{Strings}\,,\, 1 \leq i \leq D_c \}$ are the attribute-value pairs contained in the event. A is the set of attribute labels, for example, $sip, dip, etype$. D_c is the number of attributes in an event of type c. This normalization of data to events ensures that the analysis algorithms are independent of the input domain.

We believe these design decisions ensure developing abstract behavior models as first-order primitives for capturing, storing, and reusing domain expertise for the analysis of networked systems. Next we discuss the syntax of such a language.

3.3 Syntax

The language grammar for defining a behavior model ϕ as a formula, consists of five key elements as shown in Figure 1: state propositions S as atomic formulae; grouping operators '(' and ')' to define sub-formulae; logical operators and temporal operators for relating sub-formulae or atomic-formulae; the optional behavior constraints $bcon$ and operator constraints $opcon$ written within '[' and ']'; and the relational operators $relop$.

A *state proposition, S,* is an atomic formula for capturing events that satisfy specified relations between at-

$$
\begin{array}{rcll}
\phi & ::= & \text{`('}\ S \mid \phi\ \text{`)'}\ \{\mathit{bcon}\} & \\
 & \mid & \texttt{not}\ \phi & (\mathit{negation}) \\
 & \mid & \phi\ \texttt{and}\ \phi & (\mathit{logical\ and}) \\
 & \mid & \phi\ \texttt{or}\ \phi & (\mathit{logical\ or}) \\
 & \mid & \phi\ \texttt{xor}\ \phi & (\mathit{logical\ xor}) \\
 & \mid & \phi \leadsto_{(opcon)} \phi & (\mathit{leadsto}) \\
 & \mid & \Box_{(opcon)}\ \phi & (\mathit{always}) \\
 & \mid & \phi\ \texttt{olap}_{(opcon)}\ \phi & (\mathit{overlaps}) \\
 & \mid & \phi\ \texttt{dur}_{(opcon)}\ \phi & (\mathit{during}) \\
 & \mid & \phi\ \texttt{sw}_{(opcon)}\ \phi & (\mathit{startswith}) \\
 & \mid & \phi\ \texttt{ew}_{(opcon)}\ \phi & (\mathit{endswith}) \\
 & \mid & \phi\ \texttt{eq}_{(opcon)}\ \phi & (\mathit{equals}) \\
\mathit{bcon} & ::= & \text{`['}\ \{tc \mid cc\}\ \text{`]'} & \\
tc & ::= & \{\texttt{at} \mid \texttt{duration} \mid \texttt{end}\}\ \mathit{relop}\ t\{:t\} & \\
cc & ::= & \{\texttt{icount} \mid \texttt{bcount} \mid \texttt{rate}\}\ \mathit{relop}\ c\{:c\} & \\
\mathit{opcon} & ::= & \text{`['}\ \mathit{relop}\ t\{:t\}\ \text{`]'} & \\
\mathit{relop} & ::= & \{ > \mid < \mid = \mid \geq \mid \leq \mid \neq \} & \\
t & ::= & [0-9]+\ \{s|ms\} & \\
c & ::= & [0-9]+ &
\end{array}
$$

Figure 1: The grammar for specifying a behavior model ϕ.

tributes and their values. In essence, S captures states of a system or process and is the basic element of a behavior model. The most trivial behavior model is one with a single state proposition. Formally, S is represented as a finite collection of related attribute-value tuples as:

$$
S = \{(a_i, r_i, v_i) \mid i \in \mathbb{N},\ a_i \in A,\ v_i \in V,\ r_i \in (=, >, <, \geq, \leq, \neq)\}
$$

A is a set of string labels, such as `sip`, `dip`, `etype` and V is a set of string constants, such as `10.1.1.2`, `/bin/sh`, along with two special strings: (a) strings prefixed with '$', as in `$$`, `$s2.dst` (b) strings with the wild-card character '*', as in `/etc/pas*`. Considering our previous example of `IPFLOW`, the state propositions `ip_s2d` and `ip_d2s` are written as:

```
ip_s2d = {etype=PKT_IP, sip=$$,dip=$$}
ip_d2s = {etype=PKT_IP,sip=$ip_s2d.dip,
          dip=$ip_s2d.sip}
```

State proposition `ip_s2d` contains three attributes `etype`, `sip` and `dip`. `etype` has a constant value `PKT_IP`, while `sip` and `dip` attributes use the '$' prefixed special variables which are dynamically bound at runtime. State proposition `ip_d2s` defines the values of its `sip` and `dip` attributes as being dependent on values of state `ip_s2d`. Dependent attributes along with dynamic binding of values allows leaving out details like the actual IP addresses from the specification.

The *temporal operators* allow expressing temporal relationships like ordering and concurrency between one-or-more behaviors. The linear-time temporal operator $\leadsto$ (*leadsto*), written as $\sim>$, is used to express causal relationships between behaviors. The interval temporal logic operators express concurrent relationships between behaviors as either relationships: (a) between their starttimes using `sw` (*startswith*), (b) between their endtimes using `ew` (*endswith*) or (c) between their durations using `olap` (*overlap*), `eq` (*equals*) and `dur` (*during*). The $\Box$ (*always*) operator, written as [ ], allows expressing invariant behaviors. The *logical operators* `not`, `and`, `or`, `xor` are supported for logical operations over behaviors and for creating complex behaviors.

Behavior constraints allow placing additional constraints on the matching behavior instances and are specified immediately following the behavior within square brackets. Constraints and their values are related using the standard relational operators. The six behavior constraints are divided as time constraints tc and count constraints cc. Time constraints allow constraining behavior starttime using `at`, behavior endtime using `end` and behavior duration using `duration`. The time value, t, for the constraint can be specified as a single positive value or as a range. Additionally, the values can be suffixed with either 's' or 'ms' to indicate seconds or milliseconds respectively. The count constraints allow constraining number of matching behavior instances using `icount`, the size of each behavior instance using `bcount` and rate of events within a behavior instance using `rate`. *Operator constraints* allow specifying time bounds over the temporal operators thus allowing their semantics to be slightly modified. The operator constraint values are specified as a single value or a range along with a relational operator. Table 2 presents detailed semantics of operators along with behavior and operator constraints.

Expressing a behavior in the language constitutes writing sub-formulae. Behaviors are always enclosed within parenthesis '(' and ')'. Simple behaviors are constructed by relating one-or-more state propositions using operators, while complex behaviors are constructed by relating one-or-more behaviors. The grammar also allows expressing complex behaviors using recursion and we present an example in Section 5.3. Recursive definitions allow expressing looping behavior for which the loop bounds can be optionally specified using the `bcount` behavior constraint. The current grammar does not support existential and universal quantification since such a need is not clear. We explore these language extensions as part of our future work.

Writing behavior models in the framework involves additional syntax such as namespaces, headers and variables which are discussed along with the case-studies in Section 5.1 and Section 5.2. Next section presents the formal semantics of the language.

3.4 Semantics

We first define two concepts important for understanding the semantics. A *sequential log* (L) is a finite sequence of timestamped events $L = e_1, e_2, e_3, \ldots, e_n$ such that $e_i.t \leq e_j.t$, $\forall\, i < j$. A *behavior instance* B_ϕ for a behavior model ϕ is sequence or groups of events satisfying

Behavior model ψ	Meaning of ψ	L satisfies ψ ($L \models \psi$) iff
(ϕ)	ϕ is a behavior.	$\exists B_\phi \subseteq L$ and $\lvert B_\phi \rvert > 0$
S	S is a state proposition defined as $S = \{(a_1, r_1, v_1) \ldots, (a_d, r_d, v_d)\}$.	(a) $\lvert B_S \rvert > 0$, (b) $\forall e \in B_S, \forall i \in \{1, \ldots, d\}, e.a_i$ is defined and values $e.v_i$ and $S.v_i$ satisfy relation r_i.
$(\texttt{neg}\, \phi)$	Negation of behavior is true.	$L \not\models \phi$, that is, $\lvert B_\phi \rvert = 0$
$(\phi_1 \,\texttt{and}\, \phi_2)$	Both ϕ_1 and ϕ_2 are true.	$L \models \phi_1$ and $L \models \phi_2$
$(\phi_1 \,\texttt{or}\, \phi_2)$	ϕ_1 and ϕ_2 are not both false simultaneously.	$L \models \phi_1$ or $L \models \phi_1$ or satisfies both ϕ_1 and ϕ_2
$(\phi_1 \,\texttt{xor}\, \phi_2)$	Either of ϕ_1 or ϕ_2 are true but not both.	$L \models \phi_1$ or $L \models \phi_2$ but not both
$(\phi_1 \leadsto \phi_2)$	ϕ_1 leadsto ϕ_2, that is, whenever ϕ_1 is satisfied ϕ_2 will eventually be satisfied.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_2}.starttime \geq B_{\phi_1}.endtime$
$(\phi_1 \leadsto [\leq t]\, \phi_2)$	Whenever ϕ_1 is satisfied ϕ_2 will be satisfied within t time units.	(a) $L \models (\phi_1 \leadsto \phi_2)$, (b) $B_{\phi_2}.starttime \leq (B_{\phi_1}.endtime + t)$
$(\Box\, \phi)$	ϕ is always satisfied, that is, satisfied by each event.	$\forall e \in L, e \models \phi$
$(\Box[= t]\, \phi)$	ϕ is always satisfied within every consecutive interval(epoch) of t time units.	$t > 0$ and for all consecutive intervals t, $l_t \subseteq L$ and $l_t \models \phi$
$(\phi_1 \,\texttt{sw}\, \phi_2)$	ϕ_1 starts with ϕ_2.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.starttime = B_{\phi_2}.starttime$
$(\phi_1 \,\texttt{sw}[\geq t]\, \phi_2)$	ϕ_1 starts t time units after ϕ_2.	(a) $L \models (\phi_1 \,\texttt{sw}\, \phi_2)$, (b) $B_{\phi_1}.starttime \geq (B_{\phi_2}.starttime + t)$
$(\phi_1 \,\texttt{ew}\, \phi_2)$	ϕ_1 ends with ϕ_2.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.endtime = B_{\phi_2}.endtime$
$(\phi_1 \,\texttt{ew}[= t]\, \phi_2)$	ϕ_1 ends t time units after ϕ_2.	(a) $L \models (\phi_1 \,\texttt{ew}\, \phi_2)$, (b) $B_{\phi_1}.endtime = (B_{\phi_2}.endtime + t)$
$(\phi_1 \,\texttt{olap}\, \phi_2)$	ϕ_1 overlaps ϕ_2, that is, ϕ_1 starts after ϕ_2 starts but before ϕ_2 ends and ends after ϕ_2 ends.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $(B_{\phi_2}.starttime < B_{\phi_1}.starttime < B_{\phi_2}.endtime)$ and $(B_{\phi_1}.endtime > B_{\phi_2}.endtime)$
$(\phi_1 \,\texttt{olap}[> t]\, \phi_2)$	ϕ_1 overlaps ϕ_2 and the overlapping region is greater than t time units.	(a) $L \models (\phi_1 \,\texttt{olap}\, \phi_2)$, (b) the overlap $(B_{\phi_2}.endtime - B_{\phi_1}.starttime) > t$
$(\phi_1 \,\texttt{eq}\, \phi_2)$	ϕ_1 equals ϕ_2 in duration.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $B_{\phi_1}.duration = B_{\phi_2}.duration$
$(\phi_1 \,\texttt{eq}[= t]\, \phi_2)$	ϕ_1 and ϕ_2 are both of duration t.	(a) $L \models (\phi_1 \,\texttt{eq}\, \phi_2)$, (b) $B_{\phi_1}.duration = B_{\phi_2}.duration = t$
$(\phi_1 \,\texttt{dur}\, \phi_2)$	ϕ_1 occurs during ϕ_2, that is, ϕ_1 starts after ϕ_2 and ends before ϕ_2 ends.	(a) $L \models \phi_1$ and $L \models \phi_2$, (b) $B_{\phi_1}[1] \neq B_{\phi_2}[1]$, (c) $(B_{\phi_1}.starttime > B_{\phi_2}.starttime)$ and $(B_{\phi_1}.endtime < B_{\phi_2}.endtime)$
$(\phi_1 \,\texttt{dur}[= t_1 : t_2]\, \phi_2)$	ϕ_1 occurs during ϕ_2 with duration between t_1 and t_2.	(a) $L \models (\phi_1 \,\texttt{dur}\, \phi_2)$, (b) $(t_1 \leq B_{\phi_1}.duration \leq t_2)$
$(\phi)[\texttt{icount} = c]$	The number of behavior instances satisfying ϕ is c.	(a) $L \models \phi$, (b) there exist distinct $B^1_\phi \ldots B^c_\phi \subseteq L$
$(\phi)[\texttt{bcount} = c]$	Behavior instances satisfying ϕ are of size c.	(a) $L \models \phi$, (b) $B_\phi.bcount = c$
$(\phi)[\texttt{rate} > c]$	Behavior instances satisfying ϕ have a rate, defined as (behavior size / behavior duration) greater than c.	(a) $L \models \phi$, (b) $(B_\phi.bcount/B_\phi.duration) > c$ and $B_\phi.duration > 0$
$(\phi)[\texttt{at} < t]$	Starting time of behavior instances satisfying ϕ must be less than absolute time t.	(a) $L \models \phi$, (b) $B_\phi.starttime < t$
$(\phi)[\texttt{end} \geq t]$	Behavior instances satisfying ϕ have endtime greater than absolute time t.	(a) $L \models \phi$, (b) $B_\phi.endtime \geq t$
$(\phi)[\texttt{duration} \neq t]$	Behavior instances satisfying ϕ are of duration $\neq t$.	(a) $L \models \phi$, (b) $B_\phi.duration \neq t$

Table 2: Semantics of operators, behavior constraints and operator constraints in our logic. We describe semantics for constraints considering only a single relational operator and refer the reader to the framework webpage [17] for details.

the behavior model ϕ.

$$B_\phi = \langle starttime, endtime, bcount, (b_1, b_2, \ldots, b_k)\rangle$$

where $(b_1, b_2, \ldots, b_k) \subseteq L$ could be an individual event e or another behavior-instance B_{ϕ_i}. $starttime = b_1.starttime$ is the starting time of the behavior as defined by its first element and $endtime = b_k.endtime$ is the ending time of the behavior as defined by its last element. $bcount = k$ is the total number of elements in the behavior instance. All b_i's are in increasing time-order of their $starttime$. Additionally, let $B_\phi.duration$ = $(B_\phi.endtime - B_\phi.startime)$ be the duration of the behavior instance and $\lvert B_\phi \rvert = B_\phi.bcount$ represent the size of behavior instance. If ϕ is a simple behavior, such as a state proposition S, then

$$B_s = \langle e_{i_1}.t, e_{i_k}.t, k, (e_{i_1}, \ldots, e_{i_k})\rangle$$

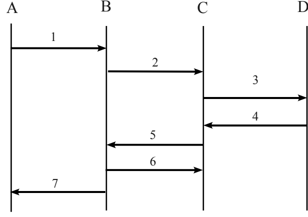


Figure 2: Sequence diagram of IP-interaction between four nodes. $\rightarrow$ or $\leftarrow$ represent an IP packet between a source (s) and destination (d). An IP flow is a packet pair between s and d.

where $(e_{i_1}, \ldots, e_{i_k}) \subseteq L$.

Given a finite sequential log L and a user-defined behavior model ϕ, goal of the analysis is to find all behavior instances $(B^1_\phi, B^2_\phi, \ldots)$ from L that satisfy the behavior model, where *satisfiability* is defined as follows:

$$L \models \phi \text{ iff } \exists B_\phi \subseteq L \text{ and } |B_\phi| > 0$$

That is, the log L satisfies ($\models$) the behavior model ϕ iff there exists a behavior instance B_ϕ in L of finite length $|B_\phi|$. Since ϕ is a composite formula created using many sub-formulas, the satisfiability of ϕ is determined as a function of satisfiability of its sub-formulae. Table 2 defines the satisfiability criteria for sub-formulae formed using the operators and constraints. We next explain the key language ideas by defining simple models and applying them to a fictitious data set.

Assume a packet trace of seven IP packets representing an interaction between four nodes A, B, C and D as shown in Figure 2. Let the sequential log of corresponding events be $e_1, e_2, \ldots, e_7$.

Using the states `ip_s2d` and `ip_d2s` defined earlier in Section 3.3, IP flow behavior is written as a causal relationship between the state propositions `ip_s2d` and `ip_d2s` as `IPFLOW=(ip_s2d ⇝ ip_d2s).` There are three IP flow instances in Figure 2 that satisfy `IPFLOW`, that is, $icount = 3$ with $bcount = 2$ for each instance:

$$B^1_{ipflow} = (e_1, e_7)$$
$$B^2_{ipflow} = (e_2, e_5)$$
$$B^3_{ipflow} = (e_3, e_4)$$

Extending the example, a complex behavior for pairs of overlapping IP flows can now be written as `IPFLOW_PAIRS=(IPFLOW olap IPFLOW).` There are in all three instances of overlapping `IPFLOW` pairs from Figure 2. That is,

$$B^1_{ipflow_pairs} = ((e_1, e_7), (e_2, e_5))$$
$$B^2_{ipflow_pairs} = ((e_1, e_7), (e_3, e_4))$$
$$B^3_{ipflow_pairs} = ((e_2, e_5), (e_3, e_4))$$

Again, $icount = 3$ and for each instance $bcount = 2$, since $bcount$ counts the number of `IPFLOW` occurrences and not individual events.

We can additionally define a bad IP flow behavior `BAD_IPFLOW` as one for which there was no matching response from the destination. That is, `BAD_IPFLOW=(ip_s2d ⇝ (not ip_d2s)).` Event e_6 matches `BAD_IPFLOW` model since it has no matching response. That is, $B^1_{bad_ipflow} = (e_6)$, with $bcount = 1$.

The next section describes the architecture of the analysis framework.

4 Semantic Analysis Framework

Given our objective of semantic-level data analysis, we require the analysis framework to support (a) analysis of multi-type, multi-variate, timestamped data, (b) defining new models by composing existing models, and (c) storage, retrieval and extensibility of domain-specific behavior models. The framework has five components as shown in Figure 3; the knowledge base, a data normalizer, an event storage system, an analysis engine and a presentation engine. The decoupling of behavior model specification, the input processing and the analysis algorithms, allows the framework to be directly applied across several different domains. Subsequent sections discuss the details of each component.

4.1 Knowledge Base

The knowledge base provides a namespace-based storage mechanism to store behavior models and is central in providing an extensible framework. For example, our networking domain currently defines models for *ipflow*, *tcpflow*, *icmpflow* and *udpflow*. These behavior models capture common domain information and allow a user to rapidly compose higher-level models by reusing existing behavior models. Reusing a behavior model from the knowledge base constitutes importing it using its namespace and name. For example, referring to the behavior model in Figure 4(a), line 5 imports the `IPFLOW` model from the `NET.BASE_PROTO` domain. The namespace allows categorization of models into domain-specific areas while allowing composition of models across domains. We implement namespaces similar to Java namespaces, that is, each component in the namespace corresponds to a directory name on the filesystem. This simple design ensures that the knowledge base is easily customizable and extensible.

4.2 Data Normalizer

The data normalizer maps a data record to the event format defined in Section 3.2. Raw data accepted by the normalizer can be in the form or trace files, packet dumps, audit logs, security logs, syslogs, kernel logs or script output with the only requirement that each data record have a timestamp and a message field. Specialized plugins in the normalizer convert each type of raw data into corresponding events. Figure 3(b) shows a possible event

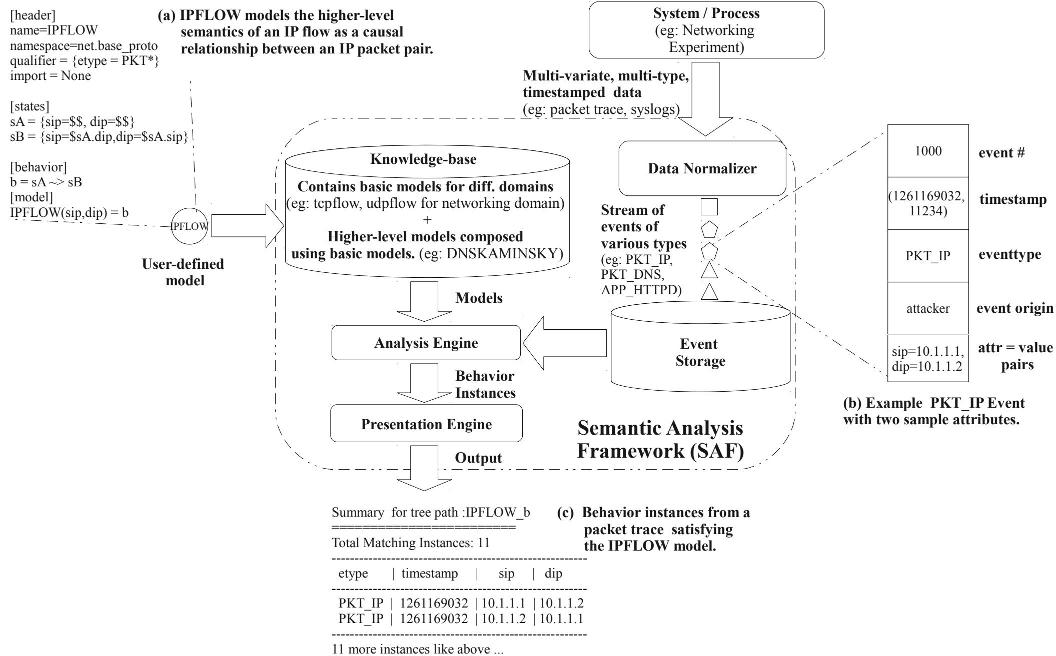


Figure 3: The semantic analysis framework (SAF) captures a user's higher-level analysis intent as (a) a behavior model, applies the model over (b) a finite stream of events normalized from raw data, and (c) outputs events satisfying the behavior model.

format for an IP packet from a packet dump. The current normalizer supports a C-based plugin API for writing new specialized plugins. The framework includes plugins for the basic packet-types of IP, TCP, UDP, ICMP, DNS along with plugins for parsing syslog, auth and server logs.

4.3 Event Storage

The event storage component is responsible for storing the events from the data normalizer into a database. Every event-type has a separate table, the columns of the tables correspond to the event attributes and each row describes an event. The current implementation stores all events into a SQLite database for two reasons: (a) it provides a standard and ready-to-use interface for storing and fetching events and (b) its server-less operation and open-source nature ensures portability on commodity systems. Our experience suggests that SQLite performs reasonably well for a large number of situations but presents challenges for complex analysis as the volume of events increases. Our future work includes investigating the scale and efficiency challenges involved in storage and retrieval of events.

4.4 Analysis and Presentation Engine

Given a finite sequential log L and a user-defined behavior model ϕ, goal of the analysis engine is to find all behavior instances $(B^1_\phi, B^2_\phi, \ldots)$ from L that satisfy the behavior model. Let the events in L be stored internally in the event storage database E_{db}. We discuss only the key ideas behind the analysis process by describing extraction of behavior instances satisfying the `IPFLOW` model defined in Section 3.4 from the sample data in Figure 2.

The behavior model ϕ is first internally represented in a manner similar to a compiler expression-tree and is then evaluated left-to-right in a post-order fashion. The satisfiability of the behavior model is determined as a function of satisfiability of each of the component behaviors according to the semantics defined in Table 2. For the `IPFLOW` model, the state proposition `ip_s2d={etype=PKT_IP,sip=$$,dip=$$}` is evaluated first. Since it does not have any dependent attributes, its expression is converted to the following query `{etype=PKT_IP,sip=*,dip=*}` and is used to fetch all events in E_{db} matching the query. All events $(e_1, e_2, e_3, e_4, e_5, e_6, e_7)$ match the state `ip_s2d`.

Next, the proposition `ip_d2s={etype=PKT_IP, sip=$ip_s2d.dip, dip=$ip_s2d.sip}` is evaluated. The attributes depend on the attributes of state `ip_s2d`. So, using each event that matched `ip_s2d`, a corresponding query is generated by resolving the values of `sip` and `dip` using the values from the matched events. From Figure 2, e_1 matches e_7, e_2 matches e_5, e_3 matches e_4. e_5 and e_6 are also possible candidates but since e_5 already matched e_2, it is not paired with e_6. Finally, the operator $\leadsto$ is evaluated, where the satisfiability criteria described in Table 2 is applied and any specified operator constraints are checked. The three instances satisfying the criteria (e_1,e_7), (e_2, e_5), and (e_3,e_4) are returned.

The *presentation engine* is responsible for extracting the output from the analysis stage and presenting it in a summarized format. We currently support printing the output in a tabular format as shown in Figure 3(c). We next present a brief analysis of the algorithm.

Algorithm Analysis As described in Section 3.3, state propositions could either contain constant attribute-values (*cStates*), such as `10.1.1.2`; dependent values (*dStates*), such as `$s1.dip`; or dynamic values (*iStates*), such as `$$`. A simple behavior consists of a combination of these states using one or more combinations of operators and constraints. We assume a constant processing time for all operators and constraints. Then, given an input of N events, processing a state proposition can involve two important operations which influence the runtime: (i) querying using the state expression and (ii) processing the results of the query if any. In the case of *cStates* and *iStates*, there is exactly one query made, and it generates at most N responses. Thus, the worst case for processing those N responses is $O(N)$. In the case of a *dstate*, given N events, there are N queries to be made and in the worst case every query may return $O(N)$ results that have to be processed. Thus, processing dependent states involves a worst case of $O(N^2)$ operations. We present our performance results in Section 6.

5 Case Studies

In this section, we evaluate the utility of our semantic framework by applying it to five different analysis scenarios: (a) confirming a hypothesis on collected network traces, (b) specifying expected system behavior during network experimentation, (c) modeling worm behavior as an example security threat, (d) modeling dynamic change, and (e) rapidly composing models to create higher-level behaviors. We present detailed explanation of input, the behavior model and analysis output for the first two cases. Due to space constraints, we briefly discuss the remaining three cases with their corresponding behavior models, demonstrating features of our semantic analysis framework.

5.1 Modeling Hypothesis

Researchers frequently need to validate hypothesis or test results presented by other researchers. We emulate one such scenario by validating the results presented by Hussain et al. [5] to demonstrate how behavior models can be rapidly created to reproduce results. We also discuss the syntax involved in writing a complete behavior model.

In the above referenced paper, a threshold-based heuristic was presented to identify DDoS attacks in traces captured at an ISP. Attacks on a victim were identified by testing for two thresholds on anonymized traces: (a) the number of sources that connect to the same destination within one second exceeds 60, or (b) the traffic

```
1. [header]
2. NAMESPACE=NET.ATTACKS
3. NAME=DDOS_HYP
4. QUALIFIER={}
5. IMPORT=NET.BASE_PROTO.IPFLOW

6. [states]
7. sA=IPFLOW.ip_s2d()
8. sB=IPFLOW.ip_s2d(dip=$sA.dip)

9. [behavior]
10.hyp_1=(sA)[bcount=1] ~>[<=1s] (sB)[bcount>=59]
11.hyp_2=(sA)[rate > 40000]

12.[model]
13.DDOS_HYP(timestamp,sip,dip,etype)= (hyp_1 or hyp_2)
```

(a) DDOS_HYP models two thresholds for detecting DDoS attacks.

```
Summary : DDOS_HYP_hyp1
=======================
Total Matching Instances: 2
Instance : 1 of 2 (Total Event Count: 60)
--------------------------------------------------------
  timestamp   |   sip        |   dip        |  etype
--------------------------------------------------------
                 State Definition: sA
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP

State Definition: ~> [<= 1 s ] sB [ ecount >= 59 ]
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP
  1025390156  |201.199.184.56|87.231.216.115| PKT_ICMP
<truncated output containing remaining 57 events>

Instance : 2 of 2 (Total Event Count: 60)
--------------------------------------------------------
  timestamp   |   sip        |   dip        |  etype
--------------------------------------------------------
                 State Definition: sA
  1025390157  |53.232.170.113|87.134.184.48 | PKT_ICMP

State Definition: ~> [<= 1 s ] sB [ ecount >= 59 ]
  1025390157  |33.138.213.170|87.134.184.48 | PKT_ICMP
  1025390157  |33.138.213.181|87.134.184.48 | PKT_ICMP
<truncated output containing remaining 57 events>
```

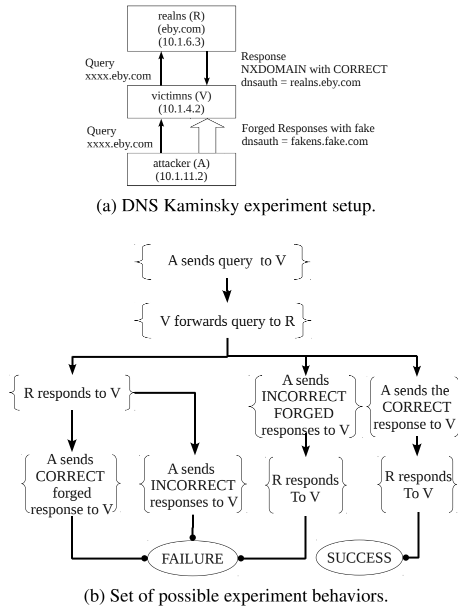
(b) Behavior instances satisfying the DDOS_HYP model.

Figure 4: Behavior model for confirming a hypothesis and corresponding behavior instances from network traces satisfying the model.

rate exceeds 40,000 packets/sec. We demonstrate the advantages of behavior model-based analysis by defining a model to test for the two heuristics listed above using 10 seconds of the trace file containing the start of an attack. We normalize the packet traces to 142,530 `PKT_IP` events.

Referring to the model script shown in Figure 4(a), lines 2–5 define the model header. Line 4 does not specify any qualifying conditions, that is, filters, for the events it can process. Line 5 imports the `IPFLOW` model from the knowledge base. Lines 7–8 define the necessary state propositions. Line 7 defines `sA`, a simple state which just captures an IP packet from some source to destination. Line 8 defines a state `sB` with a dependency that its `dip` has to be equal to the `dip` in `sA`. State `sA` thus provides a context for `sB`.

Line 10 expresses the first hypothesis that there should be more than 60 sources connecting to the same destination for an attack. We apply the $\rightsquigarrow$ operator to denote that we expect `sA` to occur before `sB`. The behavior constraint *bcount* (refer Section 3.4) applied to `sA` limits number of events returned to 1, whereas it is applied to `sB` so that atleast 59 events should occur since the event matching `sA` occurred. Additionally, the operator constraint `[<=1s]`



(a) DNS Kaminsky experiment setup.

(b) Set of possible experiment behaviors.

```
1. [header]
2. NAMESPACE = NET.ATTACKS
3. NAME = DNSKAMINSKY
4. QUALIFIER = {etype='PKT_DNS'}
5. IMPORT = NET.APP_PROTO.DNSREQRES

6. [states]
7. # Attacker to victim query
8. AtoV_query = DNSREQRES.dns_req()

9. # Victim to real ns query
10. VtoR_query= DNSREQRES.dns_req(sip=$AtoV_query.dip,
           dnsquesname=$AtoV_query.dnsquesname)

11.# Real NS to victim real response
12.RtoV_resp = DNSREQRES.dns_res($VtoR_query,
                               dnsauth=fakens.fake.com)

13.# Attacker to victim CORRECT fake response
14.AtoV_resp = DNSREQRES.dns_res($VtoR_query,
                    dnsauth=realns.eby.com) [bcount>=1]

15.# Attacker to victim INCORRECT response case
16.AtoV_noresp = DNSREQRES.dns_res($VtoR_query,
17.                dnsid != $VtoR_query.dnsid) [bcount>=1]


18.[behavior]
19.initial_query = (AtoV_query ~> VtoR_query)
20.b_1 = initial_query~>RtoV_resp ~> (AtoV_resp xor
                                    AtoV_noresp)
21.b_2 = initial_query ~> AtoV_noresp ~> RtoV_resp
22.b_3 = initial_query ~> AtoV_resp ~> RtoV_resp

23.[model]
24.FAILURE(sip,dip,sport,dport,dnsid,dnsauth) = b_1 or b_2
25.SUCCESS(sip,dip,sport,dport,dnsid,dnsauth) = b_3
```

(c) DNSKAMINSKY models complete experiment behavior.

Figure 5: Experiment setup, possible set of behaviors and corresponding behavior model for validating a networked experiment.

binds `sA` and `sB` to occur within a second in the order specified.

Line 11 defines the second hypothesis that requires that the packet rate be $\geq$ 40,000 by using the `rate` constraint on state proposition `sA`. Lastly, line 13 defines the behavior model `DDOS_HYP` which asserts that either `hyp_1` or `hyp_2` or both are valid. The four attributes `timestamp,sip,dip,etype` are reported in the final output.

When the model is applied to the packet trace, it produces an output as shown Figure 4(b). We see that there are two instances reported matching hypothesis `hyp_1` both with 60 events within a 1 second interval. The output also shows the corresponding state or behavior definitions matching the following events. The two destination IPs that are under attack are 87.231.216.115 and 87.134.184.48. This output is consistent with the findings reported in the original paper [5].

This example clearly demonstrates the ease with which simple hypotheses could be modeled and validated. The original authors wrote about 2,000 lines of C code to identify attacks. The same validation was expressed in about five lines as a behavior model. Additionally, this model can now be shared and easily modified and extended.

5.2 Modeling Experiment Behavior

Running experiments on a testbed, such as DETER [2], is challenging since it is hard to ascertain the validity of the experiment manually. With our framework, a model can be used to capture the "definition of validity" which includes possible successful and failed behaviors for an experiment and then confirmatory analysis can verify if it was met. Such a model can also be easily shared with other experimenters promoting sharing and reuse of experiments.

We present an experiment emulating Dan Kaminsky's popular DNS attack [7] using the metasploit [11] framework. Referring to Figure 5(a), the attackers objective is to poison the cache of the *victimns* so that any requests to *eby.com* are redirected to a fake nameserver (*fakens*) instead of the real nameserver (*realns*). We refer the reader to [7] for a detailed understanding of the attack. Since the attack exploits a race condition, our experiment setup has to permit successful occurrences as well as failed occurrences of the attack.

Figure 5(b) captures the experiment behavior as a tree of possibilities where the nodes are the experiment states and the paths connecting the states are possible experiment behaviors. These states are not exhaustive but sufficient to capture most of the semantics of the experiment. Specifically, we see that there are three possible behaviors that can lead to failures and one behavior that can lead to success.

The behavior model script is shown in Figure 5(c). Lines 2–4 define the model as `DNSKAMINSKY` over events of type `PKT_DNS`. Line 5 imports the `DNSREQRES` model that already defines states and behaviors relevant to the DNS protocol.

Lines 7–17 define five different states that are relevant to the experiment. Line 8 defines the first DNS query from attacker to victim and provides a context for further

```
Summary : DNSKAMINSKY_SUCESS
========================
Total Matching Instances: 1
------------------------------------------------------------------------------------------
  etype   | timestamp  | sip       |  dip      | sport | dport | dnsid |   dnsauth
------------------------------------------------------------------------------------------
  PKT_DNS | 1275515488 | 10.1.11.2 | 10.1.4.2 | 38323 |  53   | 59439 |
  PKT_DNS | 1275515488 | 10.1.4.2  | 10.1.6.3 | 32778 |  53   | 59439 |
  PKT_DNS | 1275515488 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 59439 |fakens.fakeeby.com
  PKT_DNS | 1275515488 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 59439 |realns.eby.com
------------------------------------------------------------------------------------------
Summary : DNSKAMINSKY_FAILURE
========================
Total Matching Instances: 622

<truncated output>
------------------------------------------------------------------------------------------
  etype   | timestamp  | sip       | dip      | sport | dport | dnsid |   dnsauth
------------------------------------------------------------------------------------------
  PKT_DNS | 1275515486 | 10.1.11.2 | 10.1.4.2 |  6916 |  53   | 47217 |
  PKT_DNS | 1275515486 | 10.1.4.2  | 10.1.6.3 | 32778 |  53   | 15578 |
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 15578 |realns.eby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
  PKT_DNS | 1275515486 | 10.1.6.3  | 10.1.4.2 |  53   | 32778 | 47217 |fakens.fakeeby.com
------------------------------------------------------------------------------------------
  PKT_DNS | 1275515486 | 10.1.11.2 | 10.1.4.2 | 28902 |  53   | 50921 |
<truncated output>
```

Figure 6: Behavior instances satisfying the DNSKAMINSKY model.

states. Line 10 defines a query from the victim to real nameserver by requiring that the source IP address of this query be same as the destination IP address of the previous query and the DNS questions of both states be identical. This makes sure that the forwarded query by the victim nameserver is the same as the one received. Line 12 defines the response from the real nameserver to the victim nameserver. The response is related to the request in line 10 by using the state identifier of the query state VtoR_query. To specifically distinguish this response from the attacker's response, we mention the value of the dnsauth attribute that is expected in the response. There are two cases for specifying the attacker's response. Line 14 defines the attacker's response same as the real nameserver response except that we mention the fake nameserver as value of the dnsauth attribute. Line 16 defines the case where the attacker's response is incorrect due to a wrongly guessed DNS transaction id. The bcount constraint specifies that any number of responses can be matched since the attacker can send multiple forged responses. Attribute values not defined in the above states default to their definitions in DNSREQRES.

Lines 19–22 specify four possible behaviors corresponding to the four different paths in Figure 5(b). Line 20 uses the xor operator to merge two behavior paths. The other behaviors use the $\rightsquigarrow$ operator to capture the causation between the states. Finally, the behavior model is defined in the model section using FAILURE and SUCCESS behaviors. Referring to Figure 5(b), we see that b_1 and b_2, where b_1 is a composite of two behaviors, lead to FAILURE and b_3 leads to SUCCESS. By default, the framework composes the final model by or'ing the behaviors specified in the model section.

After running the experiment and capturing DNS packets, we normalize the last 10,000 packets to PKT_DNS events since they contain a successful attack along with failures representative of rest of the capture. The framework outputs one SUCCESS instance and 622 FAILURE instances as shown in Figure 6.

```
1. scan_A     = {etype=SCAN, src=$infect_A.host, dst=$$}
2. infect_A   = {etype=INFECT, host=$scan_A.dst}
3. single_spread  = (scan_A ~> infect_A)
4. spread_chain   = (single_spread ~> spread_chain)
5. WORMSPREAD(host) = (spread_chain)
```

(a) Modeling the worm infection chain over IDS alerts.

```
1. IMPORT = NET.APP_PROTO.HTTP
2. http_pkt     = HTTP.HTTP_PKT(sip=$$, dip=$$)
3. attack_event = {etype=DOSATTACK,src=$$,dst=http_pkt.dip}
4. http_stream_at100 = ((http_pkt)[rate=100])
5. http_stream_below50  = ((http_pkt)[rate=0:50])
6. attack_start=(http_stream_at100 ew[<= 5s](attack_event))
7. DYNAMIC_CHANGE = (attack_start ~> http_stream_below50)
```

(b) Modeling change in rate of packet streams.

```
1. IMPORT = NET.ATTACKS.DNSKAMINSKY,NET.ATTACKS.WORMSPREAD
2. worm_attack= WORMSPREAD.single_spread(host=$$)
3. dns_attack = DNSKAMINSKY.SUCCESS(sip=$worm_attack.host)
4. COMBINED_ATTACK = (worm_attack ~> (dns_attack))
```

(c) Modeling an attack by composing WORMSPREAD and DNSKAMINSKY models.

Figure 7: Excerpts from behavior models for (a) modeling a security threat, (b) modeling a dynamic change and (c) composing higher-level models. We refer the reader to the framework webpage [17] for details.

This case study demonstrates the ease with which the full system behavior was semantically modeled at the level of user's understanding. Additionally, the model was composed using existing models from the knowledge base, extended with user's context-specific values for attributes and then validated.

5.3 Modeling a Security Threat

In this case study, we define a behavior model of a typical worm spread detected by IDS alerts collected from multiple hosts. Assume a network with IDSes on each host reporting two types of timestamped alerts: a SCAN alert when a scan is detected by a host and an INFECT alert when the host is found infected. Assume an event log created by normalizing the alerts to two types of events with their corresponding attributes. Given the event log, our objective here is to define a behavior model to extract all possible infection chains of any length and report the hosts involved.

We model the worm spread behavior as shown in Figure 7(a) in two stages; by first defining a single_spread behavior using events from a single host and then defining the spread_chain as a chain of related single_spread occurrences. The single_spread behavior, concerning a vulnerable host A, is a sequence of two dependent and casual events: (a) a scan_A event with its src attribute pointing to an earlier infected host, followed by (b) an infect_A event with its host attribute the same as scan_A.dst. A worm spread chain (spread_chain) is then simply defined by a recursive occurrence of related single_spread behaviors. Referring to the model, the forward-dependent attribute src in the definition of scan_A connects successive single_spread behaviors by requiring the src

of the next scan to be the same as the previously infected host. The forward-dependent attribute `src` is initialized automatically the first time `single_spread` is parsed by considering it to be a dynamic ($$) variable. The next iteration over `spread_chain` then uses the values as determined dynamically by `single_spread`.

5.4 Modeling Dynamic Change

Dynamic changes are a fundamental characteristic of networked and distributed environments. One example of a dynamic change is the change in rate of a stream of packets due to an anomalous condition such as a DoS attack. Our objective in this case study is to model an expected reduction in the rate of legitimate HTTP traffic due to DoS attack on a server. Our raw data consists of IDS DoS attack alerts and HTTP packets.

The `DYNAMIC_CHANGE` model, containing only the relevant aspects is described in Figure 7(b). Line 2 defines a state capturing a HTTP packet between a source and destination. Line 3 defines a state capturing a DoS attack alert, additionally requiring the destination to be same as the destination in the HTTP packet. Lines 4 and 5 describe the HTTP packet stream rates before and after the attack respectively. The change boundary is defined by the `attack_event` that is triggered once the attack starts. Since `attack_event` represents a single event, it has the same starttime and endtime. Line 6 use the `ew` (endswith) operator to define the `attack_start` condition, which specifies that the `http_stream_at100` behavior end within five seconds of the `attack_event`. The `DYNAMIC_CHANGE` model is then an assertion that the HTTP stream rate reduces following the attack.

5.5 Composing Models

Our final case study demonstrates the ease of composing and extending existing models to define semantically relevant higher-level behavior.

We combine our previously defined models `DNSKAMINSKY` and `WORMSPREAD` to create a `COMBINED_ATTACK` scenario as shown in Figure 7(c). Line 2 captures the behavior where a worm infects a host machine and scans and infects another host. Line 3 describes the behavior where the worm launches a DNS Kaminsky attack on some DNS server from the last infected host. We do not specify any server for the DNS Kaminsky attack due to the abstractness of the `DNSKAMINSKY` model which infers the destination dynamically. Line 4 is the final behavior model combining both the attacks. In line 3, we only constrain the `sip` and leave other attributes unspecified. This demonstrates the ability to extend the imported models with only the desired attribute values while leaving the others as defined in the imported model.

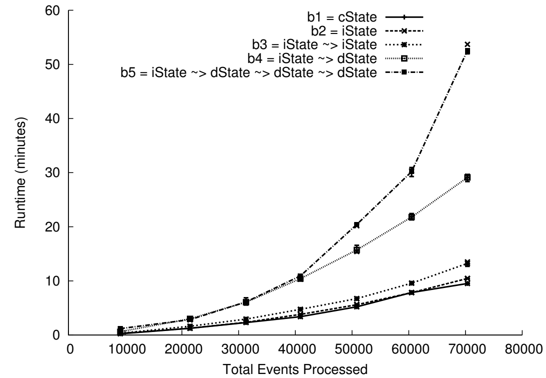


Figure 8: Plot of runtime against number of events for five types of behavior complexity. Behaviors containing dependent value states (dStates) result in quadratic complexity.

6 Performance Analysis

A common approach for semantic-level analysis involves use of custom scripts or tools encoding context-specific semantics. Since custom scripts and tools can be written using a variety of programming and optimization techniques, any evaluation of our generic framework against them would be very subjective and thus flawed. Instead, we choose to report the raw runtime performance of our prototype implementation on five basic analyses tasks over event datasets of increasing size.

The runtime performance of the framework depends on the language constructs, input data, analysis algorithm and implementation mechanisms used. Since our primary focus in this paper is on enabling semantic functionality, we prototyped the framework in Python using a SQLite database as backend for storing events. The input events used were `PKT_DNS` events collected for the case study in Section 5.2. The performance analysis was conducted on a laptop with an Intel Pentium-M processor running at 1.86 GHz and with a memory of 2 GB.

We measure runtime as a function of two variables: (a) the number of events input to the algorithm, (b) the *behavior complexity*, defined as the processing complexity of state propositions in a behavior formula. As discussed in Section 3.3, there are three types of state propositions based on attribute assignments; constant value attributes denoted as *cState*, dependent value attributes denoted as *dState*, and dynamic attribute values denoted as *iState*. These states can be combined to form five basic behaviors, each representing a basic semantic analysis task: b1 = (cState), represents extracting events with known attributes and values; b2 = (iState), represents extracting events with particular attributes but unknown values; b3 = (iState $\rightsquigarrow$ iState), represents extracting causally correlated yet value-independent events; b4 = (iState $\rightsquigarrow$ dState), represents extracting causally correlated and value-dependent events; and b5 = (iState $\rightsquigarrow$

dState ⇝ dState ⇝ dState), represents extracting a long chain of causal events. Although we limit our analysis to the ⇝ operator, all operators incur uniform processing overhead in the algorithm, thus resulting in similar performance results. The chosen event set along with the behaviors are representative of a worst-case input to the framework. We measure the performance using above behaviors over event sets in increments of 10,000 events. We stop at the event set when runtime exceeds 60 minutes.

The results are averaged over three runs and are shown in Figure 8. The plots for behaviors consisting of *cStates* and *iStates* b1, b2 and b3 tend to be linear as discussed in Section 4.4. One would expect that behavior b5, containing three *dStates* would show significantly higher runtime than behavior b4 containing only one *dState*. Both show quadratic performance, since, in a chain of dependent states, the states further in the chain process lesser events than states in front of the chain. We thus see that runtime quickly becomes quadratic given a worst-case set of events and behaviors containing dependent state propositions. The current Python and SQLite-based implementation also add penalty to the framework runtime. We investigate these issues as part of our future work.

7 Conclusion and Future Work

In this paper, we presented a behavior-based semantic analysis framework that allows the user to analyze data at a higher-level of abstraction. Typically, system experts rely on their intuition and experience to manually analyze and categorize scenarios and then hand-craft rules and patterns for analysis. Hence due to the manual and ad-hoc nature of this analysis process, there is limited extensibility and composibility of analysis strategies. In this paper we show that our approach is more systematic, can retain expert knowledge, and supports composing behaviors from existing models. We evaluated the utility of our framework against five analyses scenarios which demonstrated the ease with which a user's higher-level understanding of system operation was expressed as behavior models over data.

Our future work includes investigating the scale and efficiency issues that arise during processing large volumes of data in both offline and real-time settings like intrusion detection. We will investigate stream-based SQL query extensions [6] to improve performance. We will also investigate extending our logic with existential and universal quantifiers. Currently, our framework requires a user to either manually specify behavior models or use existing models from the knowledge base to explore data. To further exploratory analysis, we would need to alert users to interesting unanticipated behaviors. We are exploring data mining algorithms to automatically discover and compose behavior models from data.

The fundamental goal of the behavior-based semantic analysis framework is to introduce a semantic approach to data analysis in networked and distributed systems research and operations. We hope that this paper serves as a catalyst for further research on semantic data analysis.

References

[1] ALLEN, J. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM 26*, 11 (Nov. 1983), 832–843.

[2] BENZEL, T., BRADEN, R., KIM, D., NEUMAN, C., JOSEPH, A., SKLOWER, K., OSTRENGA, R., AND SCHWAB, S. Experience with DETER: A Testbed for Security Research. In *2nd Intl. Conf. on Testbeds and Research Infrastructures for the Devel. of Networks and Communities - TRIDENTCOM* (2006), p. 10.

[3] BÉRARD, B. *Systems and Software Verification: Model-checking Techniques and Tools*. Springer, 2001.

[4] ELLIS, D. R., AIKEN, J. G., ATTWOOD, K. S., AND TENAGLIA, S. D. A Behavioral Approach to Worm Detection. In *Proc. of the ACM workshop on Rapid malcode* (2004), pp. 43–53.

[5] HUSSAIN, A., HEIDEMANN, J., AND PAPADOPOULOS, C. A Framework For Classifying Denial of Service Attacks. *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Comp. Comm. - SIGCOMM* (2003), 99.

[6] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÇETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a Streaming SQL Standard. *Proc. VLDB Endow. 1* (August 2008), 1379–1390.

[7] KAMINSKY, D. Multiple DNS Implementations Vulnerable to Cache Poisoning. http://www.kb.cert.org/vuls/id/800113, 2008.

[8] KINDER, J., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. Detecting Malicious Code by Model Checking. In *Intrusion and Malware Detection and Vuln. Assessment*, K. Julisch and C. Kruegel, Eds., vol. 3548 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 174–187.

[9] LAMPORT, L. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst. 16*, 3 (1994), 872–923.

[10] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking: Language, Execution and Optimization. In *Proc. of ACM SIGMOD* (2006), pp. 97–108.

[11] Metasploit Framework Website. http://www.metasploit.com/.

[12] MIRKOVIC, J., SOLLINS, K., AND WROCLAWSKI, J. Managing the Health of Security Experiments. In *Proc. of the conf. on Cyber Security Experimentation and Test* (2008), USENIX, pp. 7:1–7:6.

[13] NALDURG, P., SEN, K., AND THATI, P. A Temporal Logic Based Framework for Intrusion Detection. In *Proc. of the 24th IFIP Intl. Conf. on Formal Tech. for Net. & Dist. Sys.* (2004).

[14] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-time. *Comput. Networks 31*, 23-24 (1999), 2435–2463.

[15] ROGER, M., AND GOUBAULT-LARRECQ, J. Log Auditing through Model-Checking. In *Proc. of the 14th IEEE Computer Security Foundations Workshop* (2001), pp. 220–236.

[16] Splunk Website. http://www.splunk.com/.

[17] Semantic Analysis Framework Website. http://thirdeye.isi.deterlab.net/.

[18] VAARANDI, R. SEC - A Lightweight Event Correlation Tool. *IEEE Workshop on IP Operations and Management* (2002), 111 – 115.

[19] Wireshark Website. http://www.wireshark.org/.

Paxos Replicated State Machines as the Basis of a High-Performance Data Store

William J. Bolosky*, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters and Peng Li
Microsoft and *Microsoft Research
{bolosky, dexterb, rhaagens, norbertk, pengli}@microsoft.com

Abstract

Conventional wisdom holds that Paxos is too expensive to use for high-volume, high-throughput, data-intensive applications. Consequently, fault-tolerant storage systems typically rely on special hardware, semantics weaker than sequential consistency, a limited update interface (such as append-only), primary-backup replication schemes that serialize all reads through the primary, clock synchronization for correctness, or some combination thereof. We demonstrate that a Paxos-based replicated state machine implementing a storage service can achieve performance close to the limits of the underlying hardware while tolerating arbitrary machine restarts, some permanent machine or disk failures and a limited set of Byzantine faults. We also compare it with two versions of primary-backup. The replicated state machine can serve as the data store for a file system or storage array. We present a novel algorithm for ensuring read consistency without logging, along with a sketch of a proof of its correctness.

1. Introduction

Replicated State Machines (RSMs) [31, 35] provide desirable semantics, with operations fully serialized and durably committed by the time a result is returned. When implemented with Paxos [20], they also tolerate arbitrary computer and process restarts and permanent stopping faults of a minority of computers, with only very weak assumptions about the underlying system--essentially that it doesn't exhibit Byzantine [22] behavior. Conventional wisdom holds that the cost of obtaining these properties is too high to make Paxos RSMs useful in practice for applications that require performance. For instance, Birman [4] writes:

> Given that it offers stronger failure guarantees, why not just insist that all multicast primitives be dynamically uniform [his term for what Paxos achieves]? … From a theory perspective, it makes sense to do precisely this. Dynamic uniformity is a simple property to formalize, and applications using a dynamically uniform multicast layer are easier to prove correct.
>
> But the bad news is that dynamic uniformity is *very costly* [emphasis his].
>
> On the other hand, there are major systems (notably Paxos…) in which … dynamic uniformity is the default. … [T]he cost is so high that the resulting applications may be unacceptably sluggish.

We argue that at least in the case of systems that are replicated over a local area network and have operations that often require using hard disks, this simply is not true. The extra message costs of Paxos over other replication techniques are overwhelmed by the roughly two orders of magnitude larger disk latency that occurs regardless of the replication model. Furthermore, while the operation serialization and commit-before-reply properties of Paxos RSMs seem to be at odds with getting good performance from disks, we show that a careful implementation can operate disks efficiently while preserving Paxos' sequential consistency. Our measurements show that a Paxos RSM that implements a virtual disk service has performance close to the limits of the underlying hardware, and better than primary-backup for a mixed read-write load.

The current state of the art involves weakened semantics, stronger assumptions about the system, restricted functionality, special hardware support or performance compromises. For example, the Google File System [13] uses append-mostly files, weakens data consistency and sacrifices efficiency on overwrites, but achieves very good performance and scale for appends and reads. Google's Paxos-based implementation [8] of the Chubby lock service [5] relies on clock synchronization to avoid stale reads and restricts its state to fit in memory; its published performance is about a fifth of ours[1]. Storage-area network (SAN) based disk systems often use special hardware

[1] Though differences in hardware limit the value of this comparison.

such as replicated battery-backed RAM to achieve fault tolerance, and are usually much more costly than ordinary computers, disks and networks. There are a number of flavors of primary-backup replication [4], but typically these systems run at the slower rate of the primary or the median backup, and may rely on (often loose) clock synchronization for correctness. Furthermore, they typically read only from the primary, which at worst wastes the read bandwidth of the backup disks and at best is unable to choose where to send reads at runtime, which can result in unnecessary interference of writes with reads. Many Byzantine-fault tolerant (BFT) [1, 9, 18] systems do not commit operations to stable storage before returning results, and so cannot tolerate system-wide power failures without losing updates. In contrast, our Paxos-based RSM runs on standard servers with directly attached disks and an ordinary Ethernet switch, makes no assumptions about clock synchronization to ensure correctness, delivers random read performance that grows nearly linearly in the number of replicas and random write performance that is limited by the performance of the disks and the size of the write reorder buffer, but is not affected by the distributed parts of the system. It performs 12%-69% better than primary-backup replication on an online transaction processing load.

The idea of an RSM is that if a computation is deterministic, then it can be made fault-tolerant by running copies of it on multiple computers and feeding the same inputs in the same order to each of the replicas. Paxos is responsible for assuring the sequence of operations. We modified the SMART [25] library (which uses Paxos) to provide a framework for implementing RSMs. SMART stored its data in SQL Server [10]; we replaced its store and log and made extensive internal changes to improve its performance, such as combining the Paxos log with the store's log. We also invented a new protocol to order reads without requiring logging or relying on time for correctness. To differentiate the original version of SMART from our improved version, we refer to the new code as SMARTER[2]. We describe the changes to SMART and provide a sketch of a correctness proof for our read protocol.

Disk-based storage systems have high operation latency (often >10ms without queuing delay) and perform much better when they're able to reorder requests so as to minimize the distance that the disk head has to travel [39]. On the face of it, this is at odds with the determinism requirements of an RSM: If two operations depend on one another, then their order of execution will determine their result. Reordering across such a dependency could in turn cause the replicas' states to diverge. We address this problem by using IO parallelism both before and after the RSM runs, but by presenting the RSM with fully serial inputs. This is loosely analogous to how out-of-order processors [37] present a sequential assembly language model while operating internally in parallel.

This paper presents Gaios[3], a reliable data store constructed as an RSM using SMARTER. Gaios can be used as a reliable disk or as a stream store (something like the i-node layer of a file system) that provides operations like create, delete, read, (over-)write, append, extend and truncate. We wrote a Windows disk driver that uses the Gaios RSM as its store, creating a small number of large streams that store the data of a virtual disk. While it is beyond the scope of this paper, one could achieve scalability in both performance and storage capacity by running multiple instances of Gaios across multiple disks and nodes.

We use both microbenchmarks and an industry standard online transaction processing (OLTP) benchmark to evaluate Gaios. We compare Gaios both to a local, directly attached disk and to two variants of primary-backup replication. We find that Gaios exposes most of the performance of the underlying hardware, and that on the OLTP load it outperforms even the best case version of primary-backup replication because SMARTER is able to direct reads away from nodes that are writing, resulting in less interference between the two.

Section 2 describes the Paxos protocol to a level of detail sufficient to understand its effects on performance. It also describes how to use Paxos to implement replicated state machines. Section 3 presents the Gaios architecture in detail, including our read algorithm and its proof sketch. Section 4 contains experimental results. Section 5 considers related work and the final section is a summary and conclusion.

2. Paxos Replicated State Machines

A state machine is a deterministic computation that takes an input and a state and produces an output and a new state. Paxos is a protocol that results in an agreement on an order of inputs among a group of replicas, even when the computers in the group crash

[2] SMART, Enhanced Revision.

[3] Gaios is the capital and main port on the Greek island of Paxos.

and restart or when a minority of computers permanently fail. By using Paxos to serialize the inputs of a state machine, the state machine can be replicated by running a copy on each of a set of computers and feeding each copy the inputs in the order determined by Paxos.

This section describes the Paxos protocol in sufficient detail to understand its performance implications. It does not attempt to be a full description, and in particular gives short shrift to the view change algorithm, which is by far the most interesting part of Paxos. Because view change happens only rarely and is inexpensive when it does, it does not have a large effect on overall system performance. Other papers [20, 21, 23] provide more in-depth descriptions of Paxos.

2.1 The Paxos Protocol

As SMART uses it, Paxos binds requests that come from clients to *slots*. Slots are sequentially numbered, starting with 1. A state machine will execute the request in slot 1, followed by that in slot 2, *etc*. When thinking about how SMART works, it is helpful to think about two separate, interacting pieces: the Agreement Engine and the Execution Engine. The Agreement Engine uses Paxos to agree on an operation sequence, but does not depend on the state machine's state. The Execution Engine consumes the agreed-upon sequence of operations, updates the state and produces replies. The Execution Engine does not depend on a quorum algorithm because its input is already linearized by the Agreement Engine.

The protocol attempts to have a single computer designated as *leader* at any one time, although it never errs regardless of how many computers simultaneously believe they are leader. We will ignore the possibility that there is not exactly one leader at any time (except in the read-only protocol proof sketch in Section 3.3.2) and refer to **the** leader, understanding that this is a simplification. Changing leaders (usually in response to a slow or failed machine) is called a *view change*. View changes are relatively lightweight; consequently, we set the view change timeout in SMART to be about 750ms and accept unnecessary view changes so that when the leader fails, the system doesn't have to be unresponsive for very long. By contrast, primary-backup replication algorithms often have to wait for a lease to expire before they can complete a view change. In order to assure correctness, the lease timeout must be greater than the maximum clock skew between the nodes.

Figure 1 shows the usual message sequence for a Paxos read/write operation, leaving out the computation and disk IO delays. When a client wants to submit a read/write request, it sends the request to the leader (getting redirected if it's wrong about the current leader). The leader receives the request, selects the lowest unused slot number and sends a *proposal* to the computers in the Paxos group, tentatively binding the request to the slot. The computers that receive the proposal write it to stable storage and then acknowledge the proposal back to the leader. When more than half of the computers in the group have written the proposal (regardless of whether the leader is among the set), it is permanently bound to the slot. The leader then informs the group members that the proposal has been decided with a commit message. The Execution Engines on the replicas process committed requests in slot number order as they become available, updating their state and generating a reply for the client. It is only necessary for one of them to send a reply, but it is permissible for several or all of them to reply. The dotted lines on the reply messages in Figure 1 indicate that only one of them is necessary.

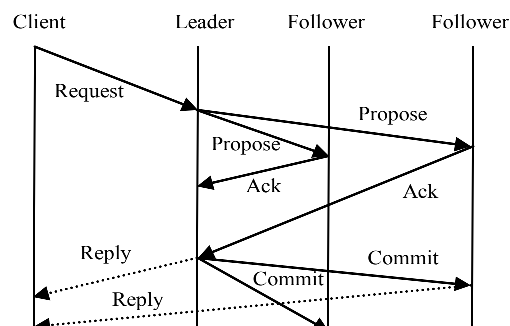


Figure 1: Read/Write Message Sequence

When the write to stable storage is done using a disk and the network is local, the disk write is the most expensive step by a large margin. Disk operations take milliseconds or even tens of milliseconds, while network messages take tens to several hundred microseconds. This observation led us to create an algorithm for read-only requests that avoids the logging step but uses the same number of network messages. It is described in section 3.3.2

2.2 Implementing a Replicated State Machine with Paxos

There are a number of complications in building an efficient replicated state machine, among them avoiding writing the state to disk on every operation. SMART and Google's later Paxos implementation [8] solve this problem by using periodic atomic checkpoints of the state. SMART (unlike Google) writes out only the changed part of the state. If a

node crashes other than immediately after a checkpoint, it will roll back its state and re-execute operations, which is harmless because the operations are deterministic. Both implementations also provide for catching up a replica by copying state from another, but that has no performance implication in normal operation and so is beyond the scope of this paper.

3. Architecture

SMARTER is at the heart of the Gaios system as shown in Figure 2. It is responsible for the Paxos protocol and overall control of the work flow in the system. One way to think of what SMARTER does is that it implements an asynchronous Remote Procedure Call (RPC) where the server (the state machine) runs on a fault-tolerant, replicated system.

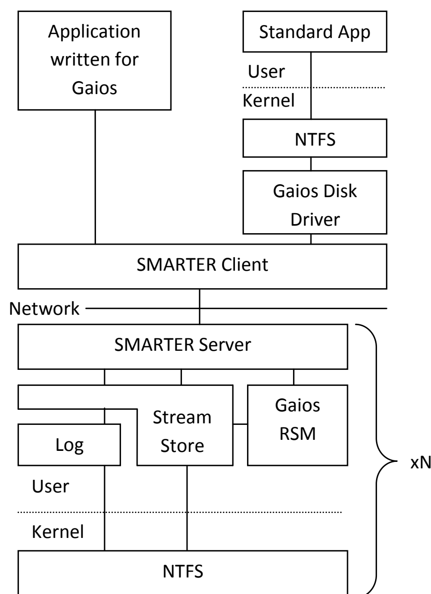


Figure 2: Gaios Architecture

Gaios's state machine implements a stream store. Streams are named by 128-bit Globally Unique IDs (GUIDs) and contain of a sparse array of bytes. The interface includes create, delete, read, write, and truncate. Reads and writes may be for a portion of a stream and include checksums of the stream data.

SMARTER uses a custom log to record Paxos proposals and the Local Stream Store (LSS) to hold state machine state and SMARTER's internal state. The system has two clients, one a user-mode library that exposes the functions of the Gaios RSM and the second a kernel-mode disk driver that presents a logical disk to Windows, and backs the disk with streams stored in the Gaios RSM.

3.1 SMARTER

Among the changes we made to SMART[4] were to present a pluggable interface for storage and log providers, rather than having SQL Server hardwired for both functions; to have a zero-copy data path; to allow IO prefetching at proposal time; to batch client operations; to have a parallel network transport and deal with the frequent message reorderings that that produces; to detect and handle some hardware errors and non-determinism; and to have a more efficient protocol for read-only requests. SMARTER performs the basic Paxos functions: client, leadership, interacting with the logging subsystem and RSM, feeding committed operations to the RSM, and managing the RSM state and sending replies to the client. It is also responsible for other functions such as view change, state transfer, log trimming, *etc*.

The SMARTER client pipelines and batches requests. Pipelining means that it can allow multiple requests to be outstanding simultaneously. In the implementation measured in this paper, the maximum pipeline depth is set to 6, although we don't believe that our results are particularly sensitive to the value. Batching means that when there are client requests waiting for a free pipeline slot, SMARTER may combine several of them into a single composite request.

Unlike in primary-backup replication systems, SMART does not require that the leader be among the majority that has logged the proposal; any majority will do. This allows the system to run at the speed of the median member (for odd sized configurations). Furthermore, there is no requirement that the majority set for different operations be the same. Nevertheless all Execution Engines will see the same binding of operations to slots and all replicas will have identical state at a given slot number.

The leader's network bandwidth could become a bottleneck when request messages are large. In this case SMARTER forwards the propose messages in a chain rather than sending them directly as shown in Figure 1. Because the sequential access bandwidth of a disk is comparable to the bandwidth of a gigabit Ethernet link, this optimization is often important.

[4] When we refer to "SMART" in the text, we mean either the original system, or to a part of SMARTER that is identical to it.

3.2 The Local Stream Store

Gaios uses a custom store called the Local Stream Store for its data (but not for its log). The LSS in turn uses a single, large file in NTFS against which it runs non-cached IO.

The LSS writes in a batch mode. It takes requests, executes them in memory, and then upon request atomically checkpoints its entire state. The LSS is designed so that it can overlap (in-memory) operation execution with most of the process of writing the checkpoint to disk, so there is only a brief pause in execution when a checkpoint is initiated.

The LSS maintains checksums for all stream data. The checksum algorithm is selectable; we used CRC32 [17] for all experiments in this paper, resulting in 4 bytes of checksum for 4K of data, or 0.1% overhead. The checksums are stored separately from the data so that all accesses to data and its associated checksum happen in separate disk IOs. This is important in the case that the disk misdirects a read or write, or leaves a write unimplemented [3]. No single misdirected or unimplemented IO will undetectably corrupt the LSS. Checksums are stored near each other and are read in batches, so few seeks are needed to read and write the checksums.

The LSS provides deterministic free space. Regardless of the order in which IOs complete and when and how often the store is checkpointed, as long as the set of requests is the same the system will report the same amount of free space. This is important for RSM determinism, and would be a real obstacle with a store like NTFS [28] that is subject to space use by external components and in any case is not deterministic in free space.

3.2.1 Minimizing Data Copies

Because SMART used SQL Server as its store, it wrote each operation to the disk four times. When logging, it wrote a proposed operation into a table and then committed the transaction. This resulted in two writes to the disk: one into SQL's transaction log and a second one to the table. The state machine state was also stored in a set of SQL tables, so any changes to the state because of the operation were likewise written to the disk twice.

For a service that had a low volume of operations this wasn't a big concern. However, for a storage service that needs to handle data rates comparable to a disk's 100 MB/s it can be a performance limitation. Eliminating one of the four copies was easy: We implemented the proposal store as a log rather than a table.

Once the extra write in the proposal phase was gone, we were left with the proposal log, the transaction log for the final location and the write into the final location. We combined the proposal log and the transaction log into a single copy of the data, but it required careful thinking to get it right. Just because an operation is proposed does not mean that it will be executed; there could be a view change and the proposal may never get quorum. Furthermore, RSMs are not required to write any data that comes in an operation—they can process it in any way they want, for example maintaining counters or storing indices, so it's not possible to get rid of the LSS's transaction log entirely.

We modified the transaction log for the LSS to allow it to contain pointers into the proposal log. When the LSS executes a write of data that was already in the proposal log, it uses a special kind of transaction log record that references the proposal log and modifies the proposal log truncation logic accordingly. The necessity for the store to see the proposal log writes is why it's shown as interposing between SMARTER and the log in Figure 2. In practice in Gaios data is written twice, to the proposal log and to the LSS's store.

It would be possible to build a system that has a single-write data path. Doing this, however, runs into a problem: Systems that do atomic updates need to have a copy of either the old or new data at all times so that an interrupted update can roll forward or backward [14]. This means that, in practice, single-write systems need to use a write-to-new store rather than an overwriting store. Because we wanted Gaios efficiently to support database loads, and because databases often optimize the on-disk layout assuming it is in-order, we chose not to build a single-write system. This choice has nothing to do with the replication algorithm (or, in fact, SMARTER). If we replaced the LSS with a log-structured or another write-to-new store we could have a single-write path.

3.3 Disk-Efficient Request Processing

State machines are defined in terms of handling a single operation at a time. Disks work best when they are presented with a number of simultaneous requests and can reorder them to minimize disk arm movements, using something like the elevator (SCAN) algorithm [12] to reduce overall time. Reconciling these requirements is the essence of getting performance from a state-machine based data store that is backed by disks.

Gaios solves this problem differently for read-only and read-write requests. Read-write requests do their writes exclusively into in-memory cache, which is cleaned in large chunks at checkpoint time in a disk-efficient order. Read-only requests (ordinarily) run on only one replica. As they arrive, they are reordered and sent to the disk in a disk efficient manner, and are executed once the disk read has completed in whatever order the reads complete.

3.3.1 Read-Write Processing

SMART's handling of read-write requests is in some ways analogous to how databases implement transactions [14]. The programming model for a state machine is ACID (atomic, consistent, isolated and durable), while the system handles the work necessary to operate the disk efficiently. In both, atomicity is achieved by logging requests, and durability by waiting for the log writes to complete before replying to the user. In both, the system retires writes to the non-log portion of the disk efficiently, and trims the log after these updates complete.

Unlike databases, however, SMART achieves isolation and consistency by executing only one request at a time in the state machine. This has two benefits: It ensures determinism across multiple replicas; and, it removes the need to take locks during execution. The price is that if two read-write operations are independent of one another, they still have to execute in the predetermined order, even if the earlier one has to block waiting for IO and the later one does not.

SMARTER exports an interface to the state machine that allows it to inspect an operation prior to execution, and to initiate any cache prefetches that might help its eventual execution. SMARTER calls this interface when it first receives a propose message. This allows the local store to overlap its prefetch with logging, waiting for quorum and any other operations serialized before the proposed operation. It is possible that a proposed operation may never reach quorum and so may never be executed. Since prefetches do not affect the system state (just what is in the cache), incorrect prefetches are harmless.

During operation execution, any reads in read/write operations are likely to hit in cache because they've been prefetched. Writes are always applied in memory. Ordinarily writes will not block, but if the system has too much dirty memory SMARTER will throttle writes until the dirty memory size is sufficiently small. The local stream store releases dirty memory as it is written out to the disk rather than waiting until the end of a flush, so write throttling does not result in a large amount of jitter.

3.3.2 Read-Only Processing

SMARTER uses five techniques to improve read-only performance: It executes a particular read-only operation on only one replica; it uses a novel agreement protocol that does not require logging; it reorders the reads into a disk-efficient schedule, subject to ordering constraints to maintain consistency; it spreads the reads among the replicas to leverage all of the disk arms; and, it tries to direct reads away from replicas whose LSS is writing a checkpoint, so that reads aren't stuck behind a queue of writes.

Since a client needs only a single reply to an operation and read-only operations do not update state there is no reason to execute them on all replicas. Instead, the leader spreads the read-only requests across the (live), non-checkpointing replicas using a round-robin algorithm. By spreading the requests across the replicas, it shares the load on the network adapters and more importantly on the disk arms. For random read loads where the limiting factor is the rate at which the disk arms are able to move there is a slightly less than linear speedup in performance as more replicas are added (see Section 4). It is sub-linear because spreading the reads over more drives reduces read density and so results in longer seeks.

When a load contains a mix of reads and writes, they will contend for the disk arm. It is usually the case that on the data disk reads are more important than writes because SMARTER acknowledges writes after they've been logged and executed, but before they've been written to the data disk by an LSS checkpoint. Because checkpoints operate over a large number of writes it is common for them to have more sequentiality than reads, and so disk scheduling will starve reads in favor of writes. SMARTER takes two steps to alleviate this problem: It tries to direct reads away from replicas that are processing checkpoints, and when it fails to do that it suspends the checkpoint writes when reads are outstanding (unless the system is starving for memory, in which case it lets the reads fend for themselves). The leader is able to direct reads away from checkpointing replicas because the replicas report whether they're in checkpoint both in their periodic status messages, and also in the MY_VIEW_IS message in the read-only protocol, described immediately hereafter.

A more interesting property of read-only operations is that to be consistent as seen by the clients, they do not need to execute in precise order with respect to the read/write operations. All that's necessary is that they execute after any read/write operation that has completed before the read-only request was issued. That is, the state against which the read is run must

reflect any operation that any client has seen as completed, but may or may not reflect any subsequent writes.

SMARTER's read-only protocol is as follows:

1. Upon receipt of a read-only request by a leader, stamp it with the greater of the highest operation number that the leader has committed in sequence and the highest operation number that the leader re-proposed when it started its view.
2. Send a WHATS_MY_VIEW message to all replicas, checking whether they have recognized a new leader.
3. Wait for at least half of all replicas (including itself) to reply that they still recognize the leader; if any do not, discard the read-only request.
4. Dispatch the read-only request to a replica, including the slot number recorded in step 1.
5. The selected replica waits for the stamped slot number to execute, and then checks to see if a new configuration has been chosen. If so, it discards the request. Otherwise, it executes it and sends the reply to the client.

In practice, SMARTER limits the traffic generated in steps 2 & 3 by only having one view check outstanding at a time, and batching all requests that arrive during a given view check to create a single subsequent view check. We'll ignore this for purposes of the proof sketch, however.

SMARTER's read-only protocol achieves the following property: The state returned by a read-only request reflects the updates made by any writes for which any client is aware of a completion at the time the read is sent, and does not depend on clock synchronization among any computers in the system. In other words, the reads are never stale, even with an asynchronous network.

We do not provide a full correctness proof for lack of space. Instead we sketch it; in particular, we ignore the possibility of a configuration change (a change in the set of nodes implementing the state machine), though we claim the protocol is correct even with configuration changes.

Proof sketch: Consider a read-only request **R** sent by a client. Let any write operation **W** be given such that **W** has been completed to some client before **R** is sent. Because **W** has completed to a client, it must have been executed by a replica. Because replicas execute all operations in order and only after they've been committed, **W** and all earlier operations must have been committed before **R** was sent. **W** was either first committed by the leader to which **R** is sent (call it **L**), or by a previous or subsequent leader (according to the total order on the Paxos view ID). If it was first committed by a previous leader, then by the Paxos view change algorithm **L** saw it as committed or re-proposed it when **L** started; if **W** was first committed by **L** then **L** was aware of it. In either case, the slot number in step 1 is greater than or equal to **W**'s slot number.

If **W** was first committed by a subsequent leader to **L**, then the subsequent leader must have existed by the time **L** received the request in step 1, because by hypothesis **W** had executed before **R** was sent. If that is the case, then by the Paxos view change algorithm a majority of computers in the group must have responded to the new view. At least one of these computers must have been in the set responding in step 3, which would cause **R** to be dropped. So, if **R** completes then **W** was not first committed by a leader subsequent to **L**. Therefore, if **R** is not discarded the slot number selected in step 1 is greater than or equal to **W**'s slot number.

In step 5, the replica executing **R** waits until the slot number from step 1 executes. Since **W** has a slot number less than or equal to that slot number, **W** executes before **R**. Because **W** was an arbitrary write that completed before **R** was started SMARTER's read-only protocol achieves the desired consistency property with respect to writes. The protocol did not refer to clocks and so does not depend on clock synchronization■

3.4 Non-Determinism

The RSM model assumes that the state machines are deterministic, which implies that the state machine code must avoid things like relying on wall clock time. However, there are sources of non-determinism other than coding errors in the RSM. Ordinary programming issues like memory allocation failures as well as hardware faults such as detected or undetected data corruptions in the disk [3], network, or memory systems [30, 36] can cause replicas to misbehave and diverge.

Divergent RSMs can lead to inconsistencies exposed to the user of the system. These problems are a subset of the general class of Byzantine faults [22], and could be handled by using a Byzantine-fault-tolerant replication system [7]. However, such systems require more nodes to tolerate a given number of faults (at least $3f+1$ nodes for f faults, as opposed to $2f+1$ for Paxos [26]), and also use more network communication. We have chosen instead to anticipate a set

of common Byzantine faults, detect them and turn them into either harmless system restarts or to stopping failures. The efficacy of this technique depends on how well we anticipate the classes of failures as well as our ability to detect and handle them. It also relies on external security measures to prevent malefactors from compromising the machines running the service (which we assume and do not discuss further).

Memory allocation failures are a source of nondeterminism. Rather than trying to force all replicas to fail allocations deterministically, SMART simply induces a process exit and restart, which leverages the fault tolerance to handle the entire range of allocation problems.

In most cases, network data corruptions are fairly straightforward to handle. SMARTER verifies the integrity of a message when it arrives, and drops it if it fails the test. Since Paxos is designed to handle lost messages this may result in a timeout and retry of the original (presumably uncorrupted) message send. In a system with fewer than f failed components, many messages are redundant and so do not even require a retransmission. As long as network corruptions are rare, message drops have little performance impact. As an optimization, SMARTER does not compute checksums over the data portion of a client request or proposal message. Instead, it calls the RSM to verify the integrity of these messages. If the RSM maintains checksums to be stored along with the data on disk (as does Gaios), then it can use these checksums and save the expense of having them computed, transported and then discarded by the lower-level SMARTER code.

Data corruptions on disk are detected either by the disk itself or by the LSS's checksum facility as described in Section 3.2. SMARTER handles a detected, uncorrectable error by retrying it and if that fails declaring a permanent failure of a replica and rebuilding it by changing the configuration of the group. See the SMART paper [25] for details of configuration change.

In-memory corruptions can result in a multitude of problems, and Gaios deals with a subset of them by converting them into process restarts. Because Gaios is a store, most of its memory holds the contents of the store, either in the form of in-process write requests or of cache. Therefore, we expect at least those memory corruptions that are due to hardware faults to be more likely to affect the store contents than program state. These corruptions will be detected as the corrupted data fails verification on the disk and/or network paths.

4. Experiments

We ran experiments to compare Gaios to three different alternatives: a locally attached disk and two versions of primary-backup replication. We ran microbenchmarks to tease out the performance differences for specific homogeneous loads and an industry standard online transaction processing benchmark to show a more realistic mixed read/write load. We found that SMARTER's ability to vector reads away from checkpointing (writing) replicas conveyed a performance advantage over primary-backup replication.

4.1 Hardware Configuration

We ran experiments on a set of computers connected by a Cisco Catalyst 3560G gigabit Ethernet switch. The switch bandwidth is large enough that it was not a factor in any of the tests.

The computers had three hardware configurations. Three computers ("old servers") had 2 dual core AMD Opteron 2216 processors running at 2.4 GHz, 8 GB of DRAM, four Western Digital WD7500AYYS 7200 RPM disk drives (as well as a boot drive not used during the tests), and a dual port NVIDIA nForce network adapter, with both ports connected to the same switch. A fourth ("client") had the same hardware configuration except that it had two quad-core AMD Opteron 2350 processors running at 2.0 GHz. The remaining two ("new servers") had 2 quad-core AMD Opteron 2382 2.6 GHz processors, 16 GB of DRAM, four Western Digital WS1002FBYS 7200 RPM 1 TB disk drives, and two dual port Intel gigabit Ethernet adapters. All of the machines ran Windows Server 2008 R2, Enterprise Edition. We ran the servers with a 128 MB memory cache and a dirty memory limit of 512 MB. We used such artificially low limits so that we could hit full-cache more quickly so that our tests didn't take as long to run, and so that read-cache hits didn't have a large effect on our microbenchmarks.

4.2 Simulating Primary-Backup

In order to compare Gaios to a primary-backup (P-B) replication system, we modified SMARTER in three ways:

1. Reads are dispatched without the quorum check in the SMARTER read protocol, on the assumption that a leasing mechanism

would accomplish the same thing without the messages.

2. Read/Write operation quorums must include the leader, so for example in a 3-node configuration if the two non-leader nodes finish their logging first the system will still wait for the leader.
3. All read/write replies come only from the leader.

Because we didn't implement a leasing mechanism, the modified SMARTER might serve stale reads after a view change. We simply ignored this possibility for performance testing.

Because P-B systems read only from the primary, they cannot take advantage of the random read performance of their backup nodes. The consequences of this may be limited by having many replication groups that spread primary duties (and thus read load) over all of the nodes. In the best case, they will uniformly spread their reads over all of the nodes as SMARTER does.

To capture the range of possible read spreading in P-B systems we implemented two versions: worst and best cases. The worst case version is called PB1 because it reads from only one node. It assumes that spreading is completely ineffective and sends all reads to the primary. The best case is called PBN and simulates perfect spreading by sending reads to all N nodes. Rather than implementing multiple groups, we simply used SMARTER's existing read distribution algorithm, but without the quorum check and without the check to avoid sending reads to nodes that are checkpointing.

The latter point is the crucial difference between the two systems. While PBN is able to use all of the disk arms for reads, it can't dynamically select which arm to use for a particular read because it must send reads to the primary, and it achieves spreading only by distributing the work of the primaries for many groups. Moving a primary is far too heavy-weight to do on each read. SMARTER, on the other hand, tries to move reads away from checkpointing replicas so that writes don't interfere with reads. It also adds some randomness into the decision about when to checkpoint to avoid having replicas checkpoint in lockstep. In the mixed read/write transaction processing load measured in section 4.4 Gaios achieves 12% better performance tan PBN because of this ability (and is 68% faster than PB1).

4.3 Microbenchmarks

We ran microbenchmarks on Gaios and P-B replication as well as directly on an instance of each of the two types of disks used in our servers, varying the number of servers from 1 to 5. We expect that most applications would want to run with a group size of 3, though a requirement for greater fault tolerance or improved read performance argues for more replicas. In all of the experiments where we varied the degree of replication, we used the three old servers first followed by the two new servers, so for instance the 4 replica data point has three old and one new server.

We used the sqlio [33] tool running on NTFS over the Gaios disk driver (or directly on the local drive, as appropriate). Gaios exported a 20 GB drive to NTFS and sqlio used a 10GB file. Gaios used two identical drives on each replica, one for log and one for the data store. Each data point is the mean of 10 measurements and was taken over a five minute period, other than the burst writes shown in Figure 4, which ran for 10 seconds. We ran all tests with the disks set to write through their cache, so all writes are durable. We ran the P-B variants only on two or more nodes because they're identical to Gaios on one node, and we ran only one P-B variant on the write tests, since PB1 and PBN differ only for reads.

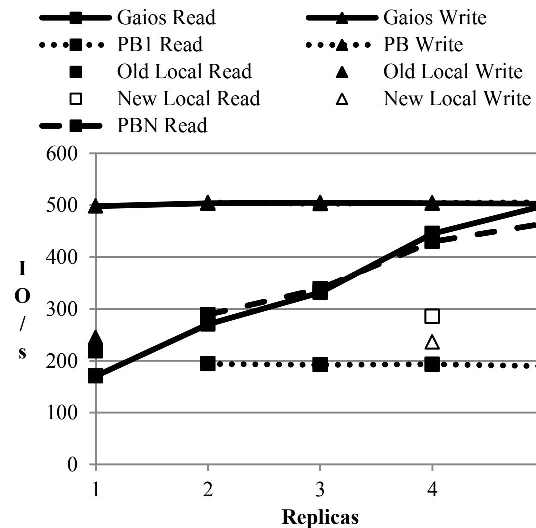


Figure 3: Random IO Performance

Figure 3 shows the performance of 8 kilobyte random reads and writes. In this and the other microbenchmark figures, we show the results for the new server disks at the 4 replica position both to provide visual separation from the old replica disks and to help point out that at 4 replicas we started adding new servers to the mix.

The writes were measured with a dirty cache. Write performance does not vary much with degree of replication or Gaios vs. P-B and is roughly 500 IO/s, a little more than twice the local disk's. This is because the server is able to reorder the writes in a disk-efficient manner over its 512MB of write buffer without the possibility of loss because the data is already logged, while the raw disks can reorder only over the simultaneously outstanding operations. The overhead of replication and checkpoints is negligible compared to disk latency, and performance is increased by SMARTER's batching.

A simple back-of-the-envelope computation shows how fast we expect the disk to be able to retire random writes, and demonstrates that SMARTER achieves that bound, meaning that (at least for random writes) the bottleneck is at the disk, not elsewhere. The disks we used have tracks about ¾ of a megabyte in size, so the 10GB sqlio file was around 14K tracks. SMARTER is using 512MB of cache, which is 64K 8KB-sized individual writes, or about 4.7 writes/track. The 7200 RPM disk takes 8.3ms for a complete rotation. 4.7 writes per each 8.3ms rotation is about 570 writes/s, which is just a little more than Gaios' performance.

The random read test used 35 simultaneous outstanding reads. Gaios' and PBN's random reads (also shown in Figure 3) scale slightly sub-linearly with the number of replicas. They improve with the number of replicas because SMARTER is able to employ the disk arms on the replicas separately, but the improvement is less than linear because as it scales each replica has fewer simultaneous reads over which to reorder. Single replica Gaios has a read rate about 14% lower than the local disk. PB1 didn't vary in the count of replicas since it only reads from one node.

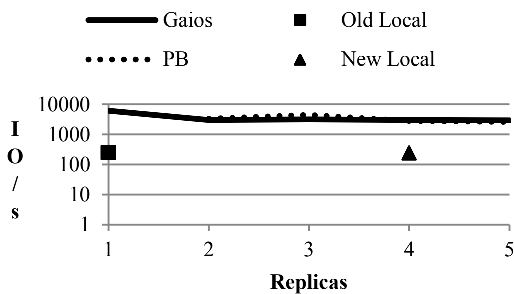


Figure 4: Burst Write Performance

Figure 4 shows the write rates for 10 second bursts of 8K random writes with 200 writes outstanding at a time. In this test, Gaios and PB logged and executed the writes and returned the replies to the client, but because the volume of data written was smaller than the 512MB dirty cache limit, it was bounded only by logging not by the seek rate of the data disk. Because SMARTER answers writes when they're written to the log, it does random write bursts at the rate of sequential writes, while the local disk does them at the rate of random writes.

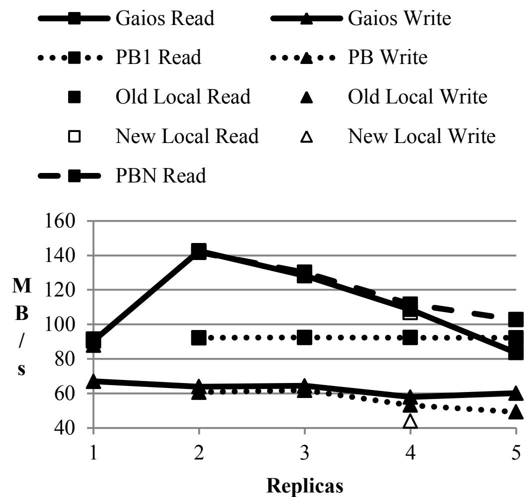


Figure 5: Sequential Bandwidth

Figure 5 shows Gaios' performance for sequential IO. This test used megabyte size requests with 40 simultaneously outstanding for writes and 10 eight megabyte requests for reads. It's difficult to see on the graph, but the (old) local disk writes at about 88 MB/s, while Gaios is at 67 MB/s. The difference is due to a difficulty in getting the data through the network transport. Writes for both Gaios and PB slow down marginally as they're distributed across more nodes (and as they need to write the slower new disks at 4 and 5 replicas). PBN and Gaios' reads are more interesting: unlike random IO, sequential IO is harder to parallelize because distributing sequential IO requests adds seeks, which reduces efficiency, sometimes more than the increase in bandwidth that's achieved by adding extra hardware. This shows up in the PBN and Gaios lines, which perform at the local disk rate on a single replica, peak at 2 replicas (but at only 1.3 times the rate of a local disk) and drop off roughly linearly after. SMARTER probably would benefit from getting hints from the RSM about how to distribute reads.

Figure 6 shows the operation latency for 8K reads and writes. Unlike the other microbenchmarks, this test only allowed a single operation to be outstanding at a time. For reads, Gaios is about 8% slower than a

local disk in the single replica case and 20% slower for 2-3 replicas. The difference in going from one to two replicas is that there is extra network traffic in the server to execute the read-only algorithm (see Section 3.3.2). Both versions of PB are about 2% faster than Gaios at 2 nodes, and 10-15% faster at 5 (where Gaios has to touch three nodes for its quorum check).

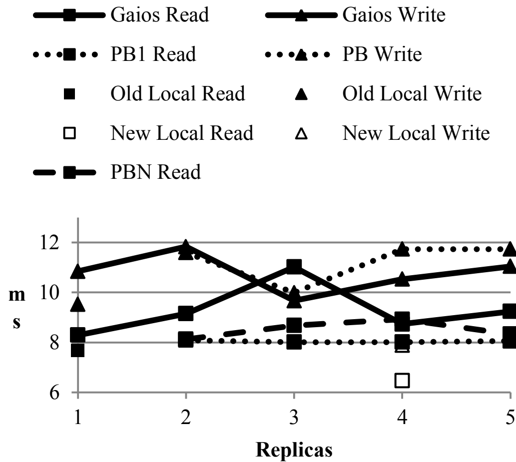


Figure 6: Single Operation Latency

Write latency is more interesting. In Gaios and P-B, the main contributor to latency is writing into the log, because the write rate is slow enough that the system doesn't throttle behind the replica checkpoint even for a 5 minute run. Writing one item to the log, waiting a little while and the writing again causes the log disk to have to take an entire 8.3ms rotation before being able to write the next log record, which accounts for the bulk of the time in Gaios. Latency goes down at three replicas because only 2 of three of them need to complete their log write for the operation to complete. As the replication grows PB gets slower than Gaios because of its requirement that the primary always be in every quorum.

The reason for storing data in an RSM is to achieve fault tolerance. To measure how Gaios performs when a fault occurs we ran a 60 second version of the 3 replica sequential read test and induced the failure of a replica half way through each of the runs. The resultant bandwidth was 127 MB/s, roughly equivalent to the 128MB/s of the non-faulty three node case. However, the maximum operation latency increased from 1500ms to 1960ms, because requests outstanding at the time of the failure had to time out and be retried. The large max latency in the non-failure case was due to the disk scheduling algorithm starving one request for a while and because of queuing delay (which is substantial with 10 8MB reads simultaneously outstanding).

4.4 Transaction Processing

In order to observe Gaios in a more realistic setting (and with a mixed read/write load), we ran an industry standard online transaction processing (OLTP) benchmark that simulates an order-entry load. We selected the parameters of the benchmark and configured the database so that it has about a 3GB log file and a 53GB table file. We housed the log and tables on different disks. In Gaios (and P-B) we ran each virtual disk as a separate instance of Gaios sharing server nodes, but using distinct data disks on the server. SMARTER shared a single log disk, so each server node used three disks: the SMARTER log, the SQL log and the SQL tables.

This benchmark does a large number of small transactions of several different types, and generates a load of about 51% reads and 49% writes to the table file by operation count, with the average read size about 9K and the average write about 10K. We configured the benchmark to offer enough load that it was IO bound. The CPU load on the client machine running SQL Server was negligible.

We used 64-bit Microsoft SQL Server 2008 Enterprise Edition for the database engine. For each data point, we started by restoring the database from a backup, which resulted in identical in-file layout. We then ran the benchmark for three hours, discarded the result from the first hour in order to avoid ramp-up effects and used the transaction rate for the second two hours. This benchmark is sensitive to two things: write latency to the SQL Server log, and read latency to the table file. The writes are offered nearly continuously as SQL Server writes out its checkpoints and are mixed with the reads.

Even though the load is half writes, the replicas spent significantly less than half of their time writing. This is because the writes were more sequential than the reads because they came from SQL's database cleaner which tries to generate sequential writes, and they were further grouped by SMARTER's checkpoint mechanism. Because of this, Gaios usually had one or more replicas that were not in checkpoint to which to send reads. Even though the load at the client was about half reads and half writes, at the server nodes it was ¾ writes because each write ran on all three nodes, while reads ran only on one. This limited the effect of the increased random read performance of Gaios and PBN.

Figure 7 shows the performance of Gaios and the two PB versions running on a three node system in transactions per second normalized to the local-machine performance. Each bar is the mean of ten runs. Gaios runs a little faster than the local node because its increased random read performance more than compensates for the added network latency and checksum IO. Because PBN is unable to direct its reads away from checkpointing nodes it is somewhat slower, while PB1 suffers even more due to its inability to extract read parallelism.

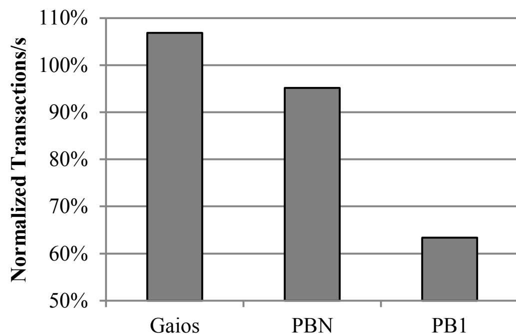


Figure 7: OLTP Performance

5. Related Work

Google [8] used a Paxos replicated state machine to re-implement the Chubby [5] lock service. They found that it provided adequate performance for their load of small updates to a state that was small enough to fit in memory (100MB). It serviced all reads from the leader (there being no need to take advantage of parallel disk access because of in-memory state), and used a time-based leasing protocol to prevent stale reads, similar to primary-backup. Their highest reported update rate was 640 small operations per second and 949 KB/s on a five node configuration, about one fifth and one sixtieth respectively of Gaios' comparable performance on 5 nodes, though because the hardware used was different it's not clear how meaningful this comparison is.

Petal [24] was a distributed disk system from DEC SRC that used two-copy primary-backup replication to implement reliability. It used a Paxos-based RSM to determine group membership, but not for data. Data writes happened in two phases, first taking a lock on the data and then writing to both copies. Only when the writes to both copies completed was the lock released and the operation completed to the user. Much like Gaios, Petal used write-ahead logging and group commit to achieve good random write performance. Castro and Liskov [7] implemented a version of NFS that stored all of its data in a BFT replicated state machine. However, their only performance evaluation was with the Andrew Benchmark [16], which has been shown [38] to be largely insensitive to underlying file system performance. BFT replication differs from Paxos in that it tolerates arbitrary, potentially malicious failures of less than a third of its replicas. It uses many more messages and a number of cryptographic operations to achieve this property.

Several BFT agreement protocols [1, 9, 18] have much lower latency than Gaios. They achieve this by not logging operations before executing them and returning results to the client. Because of this, these systems cannot tolerate simultaneous crashes of too many nodes (such as would be caused by a datacenter power failure) without permanently failing or rolling back state. As such, they do not provide sufficiently tight semantics to implement tasks that require write through such as the store for a traditional database. They also are not evaluated on state that is larger than memory. Furthermore, because they tolerate general Byzantine faults, they need at least $3f+1$ (and sometimes more) replicas to tolerate f faults (though f of these replicas can be witnesses that do not hold execution state [40]). Gaios tolerates many non-malicious (hardware or programming-error caused) Byzantine faults without the extra complexity of dealing with peers that are trying to corrupt the system.

The Federated Array of Bricks (FAB) [34] built a store out of a set of industry-standard computers and disks, much like Gaios. It used a pair of custom replication algorithms, one for mirrored data and one for erasure-coded. Unlike Paxos, it did not have a leader function or views; rather (in the mirroring case), it took a write lock over a range of bytes using a majority algorithm. Once the write lock was taken, it sent the write data to all nodes, and updated both the data and a timestamp. After a majority of the nodes completed the write, it completed the operation back to the caller. To read data, it sent the read to all replicas, with one designated to return the data. The other nodes returned only timestamps; if the returned data did not have the latest timestamp, it retried the read. This scheme achieves serializability without needing to achieve a total order of operations as happens in an RSM. However, because its read algorithm requires accessing a per-block timestamp, it employed NVRAM to avoid the need to move the disk arms to read the timestamps; SMARTER's algorithm simply asks for a copy of in-memory state from all of the replicas, and does the disk IO on only one and so does not need NVRAM.

Oceanstore [19] was designed to store the entire world's data. It modified objects by generating updates locally and then running conflict resolution in the background, in the style of Bayou [11]. Oceanstore used a Byzantine-agreement protocol to serialize and run conflict resolution, but stored the data using simple lazy replication (or replication of erasure coded data).

The Google File System [13] is designed to hold very large files that are mostly written via appends and accessed sequentially via reads. It relaxes traditional file system consistency guarantees in order to improve performance. In particular, write operations that fail because of system problems can leave files in an "inconsistent" state, meaning that the values returned by reads depend on which replica services the read. Furthermore, concurrent writes can leave file regions in an "undefined" state, where the result is not consistent with any serialization of the writes, but rather is a mixture of parts of different writes. After a period of time, the system will correct these problems. GFS uses write-to-all, so faults require the system to reconfigure before writes can proceed.

Berkeley's xFS [2] and Zebra [15] file systems placed a log structured file system [32] on top of a network RAID. They worked by doing write-to-all on the RAID stripes, and then using a manager to configure out failed storage nodes. The xFS prototype described in the paper did not "implement the consensus algorithm needed to dynamically reconfigure manager maps and stripe group maps."

Boxwood [27] offered a set of storage primitives at a higher level than the traditional array of blocks, such as B-trees. It used Paxos only to "store global system state such as the number of machines."

Everest [29] is a system that offloads work from busy disks to smooth out peak loads. When off-loading, it writes multiple copies of data to any stores it can find and keeps track of where they are in volatile memory. After a crash and restart, the client scans all of the stores to find the most up-to-date writes, and as long as one copy of each write is available, it recovers. This protocol works because there is only ever one client for a particular set of data.

TickerTAIP [6] was a parallel RAID system that distributed the function of the RAID controller in order to tolerate faults in the controller. It used two-phase commit [14] to ensure atomicity of updates to the RAID stripes.

6. Summary and Conclusion

Conventional wisdom holds that while Paxos has theoretically desirable consistency properties, it is too expensive to use for applications that require performance. We argue that compared to disk access latencies, the overhead required by Paxos on local networks is trivial and so the conventional wisdom is incorrect. While replicated state machines' in-order requirement seems to be at odds with the necessity of doing disk operation scheduling, careful engineering can preserve both.

We presented Gaios, a system that provides a virtual disk implemented as a Paxos RSM. Gaios achieves performance comparable to the limits of the hardware on which it's implemented on various microbenchmarks and the OLTP load, while providing tolerance of arbitrary machine restarts, a sufficiently small set of permanent stopping failures and some types of Byzantine failures. We compared Gaios to primary-backup replication and found that it performs comparable to or in some cases better than P-B's best case. We presented a novel read-only algorithm for SMARTER, and showed that because it allows reads to run on any node SMARTER can often avoid having reads and writes contend for a particular disk, giving significant performance improvements over even the best case of primary-backup replication for the mixed read/write workload of the OTLP benchmark.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, 2005.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang. Serverless network file systems. In *Proc. SOSP*, 1995.

[3] L. Bairavasundaram, G. Goodson, B. Schroeder, A. Arpaci-Dusseau and R. Arpaci-Dusseau. An analysis of data corruption in the storage stack. In *Proc. FAST*, 2008.

[4] K. Birman. *Reliable Distributed Systems Technologies, Web Services and Applications*. Springer, 2005

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI*, 2006.

[6] P. Cao, S. Lim, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. In *Proc. ISCA*, 1993.

[7] M. Castro and B. Liskov, Practical Byzantine fault tolerance. In *Proc. OSDI*, 1999.
[8] T. Chandra, R. Griesemer and J. Redstone. Paxos made live: an engineering perspective. In *Proc. PODC*, 2007. Invited talk.
[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shira. HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI*, 2006.
[10] K. Delaney, P. Randal, K. Tripp and C. Cunningham. *Microsoft SQL Server 2008 Internals*. Microsoft Press, 2009.
[11] A. Demers, K. Peterson, M. Spreitzer, D. Terry, M. Theimer and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proc. IEEE Workshop on Mobile Computing Systems & Applications*, 1994.
[12] R. Eager and A. Lister. *Fundamentals of Operating Systems*. Springer-Verlag, 1995.
[13] S. Ghemawat, H. Gobioff and S-T. Leung. The Google file system. In *Proc. SOSP*, 2003.
[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.
[15] J. Harman and J. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3), 1995.
[16] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1), 1988.
[17] IEEE 802.3 Standard, 1983-2008.
[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proc. SOSP*, 2007.
[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, 2000.
[20] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems,* 16(2), 1998.
[21] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.
[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*. 4(3), 1982.
[23] B. Lampson. The ABCD's of Paxos. In *Proc. PODC,* 2001.
[24] E. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proc. ASPLOS*, 1996.
[25] J. Lorch, A. Adya, W. Bolosky, R. Chaiken, J. Douceur and J. Howell. The SMART way to migrate replicated stateful services. In *Proc. Eurosys*, 2006.
[26] N. Lynch. *Distributed Algorithms*. Morgan Kauffman, 1996.
[27] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proc. OSDI*, 2004.
[28] R. Nagar. *Windows NT File System Internals*. O'Reilly, 1997.
[29] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling down peak loads through I/O off-loading. In *Proc. OSDI*, December, 2008.
[30] E. Nightingale, J. Douceur and V. Orgovan. Cycles, Cells and Platters: An empirical analysis of hardware failures on a million commodity PCs. To appear in *Proc. EuroSys,* 2011.
[31] B. Oki. Viewstamped replication for highly available distributed systems. Ph.D. thesis. Technical Report MIT/LCS/TR-423, MIT, 1988.
[32] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.
[33] M. Ruthruff. SQL Server best practices article: predeployment I/O best practices. In IEEE Computer, 27(3), 1994.
[34] Y. Saito, S. Frølund, A. Veitch, A. Merchant and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. ASPLOS*, 2004.
[35] F. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), 1990.
[36] B. Schroeder, E. Pinheiro, and W-D. Weber. DRAM Errors in the wild: A large-scale field study. In *Proc. SIGMETRICS/Performance*, 2009.
[37] R. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal of Research and Development*, 11(1), 1967.
[38] A. Traeger, E. Zadok, N. Joukov and C. Wright. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage*, 4(2), 2008.
[39] B. Worthington, G. Ganger, and Y. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proc. SIGMETRICS*, 1994.
[40] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, 2003.

Bootstrapping Accountability in the Internet We Have

Ang Li Xin Liu Xiaowei Yang
Dept. of Computer Science
Duke University
{angl,xinl,xwy}@cs.duke.edu

Abstract

Lack of accountability makes the Internet vulnerable to numerous attacks, including prefix hijacking, route forgery, source address spoofing, and DoS flooding attacks. This paper aims to bring accountability to the Internet with low-cost and deployable enhancements. We present IPA, a design that uses the readily available top-level DNSSEC infrastructure and BGP to bootstrap accountability. We show how IPA enables a suite of security modules that can combat various network-layer attacks. Our evaluation shows that IPA introduces modest overhead and is gradually deployable. We also discuss how the design incentivizes early adoption.

1 Introduction

Accountability, the ability to identify misbehaving entities and deter them from misbehaving further, plays a critical role in achieving real-world security [41]. However, the Internet design has little built-in accountability: malicious hosts can send denial of service (DoS) flooding packets with spoofed source addresses to evade punishment; and malicious Autonomous Systems (ASes) can announce other ASes' IP prefixes or assume their identities in the inter-domain routing system BGP.

Lack of accountability has led to many of the Internet's security vulnerabilities [20, 58], including distributed DoS attacks that may disable a country's Internet access [48, 49, 52], and prefix hijacking attacks that once made YouTube worldwide unreachable [25]. In this work, we ask the question: *can we overcome the Internet's main security weaknesses with a minimal set of gradually deployable changes?* That is, we aim to explore an approach that can fix the Internet's security problems without replacing or breaking the deployed Internet base. We are attracted to this approach because of its practical value, as it can deliver benefits without building everything from scratch.

In this paper, we present a design called IPA (IP made Accountable) that bootstraps accountability in the Internet with only low-cost and gradually deployable enhancements. We show how the IPA design enables other security modules that together fix many of the Internet's security problems, including preventing prefix hijacking, route forgery, and source address spoofing attacks, and limiting large-scale DoS attacks. We note that this work does not aim to provide all forms of accountability. For instance, IPA does not provide the type of strong accountability that offers evidence of correct execution, or audit and challenge interfaces [32, 60]. Rather, it aims to bring a similar form of network-layer accountability as defined in [20, 54] to the Internet, *i.e.*, the ability to accurately identify the sources of all traffic and defend against malicious sources.

We identify two key challenges in bootstrapping accountability in the existing Internet. The first one is how to securely bind an entity's identity to its cryptographic keys in a lightweight manner, and the second one is how to do so in an adoptable manner, including being gradually deployable and incentivizing early adoption. Network-layer accountability requires a secure binding between an entity's identity and its cryptographic keys to prevent impersonation and identity white-washing attacks [31]. The Internet uses two types of identifiers, IP addresses and AS numbers (ASNs), to identify network attachment points and ASes, but it lacks a lightweight and adoptable mechanism to create the secure bindings between an IP address (or an ASN) and a network entity. Previous work [38, 46, 56, 57] proposes to use a centralized global public key infrastructure (PKI) or web-of-trust to bind an IP prefix or an ASN to its owner's public key. However, a dedicated PKI is too heavyweight [35], and web-of-trust lacks an authoritative trust chain to resolve conflicting IP prefix or ASN claims.

IPA uses three mechanisms to address these challenges. First, it uses the top-level reverse DNSSEC hierarchy as a lightweight PKI to bind an IP prefix to its owner's public key (§ 3.2), and the hash of an AS's public key as its self-certifying ASN (§ 3.1). This design securely certifies an IP prefix's ownership without a separate PKI, and obviates another PKI to certify an ASN's ownership. We use DNSSEC [21, 22, 23, 50] because one can create a one-to-one mapping between an IP prefix delegation and a reverse DNS zone delegation, as the chains of trust in both delegation processes share the same root: the Internet Assigned Number Authority (IANA). Thus, we can use an IP prefix's corresponding reverse DNSSEC record as its owner's IP prefix delegation certificate. Moreover, Internet registries are rapidly deploying the top-level reverse DNSSEC infrastructure [4, 6, 18, 19]. The root, the `arpa`, and the `in-addr.arpa` zones are already signed. Deployment documents from key Regional Internet Registries (RIRs) [1, 2, 5] all suggest that the top-level reverse DNSSEC infrastructure would soon be fully deployed.

Second, IPA uses an efficient in-band protocol piggybacked in BGP messages to "push" the IP prefix certificates to all ASes to secure routing (§ 3.3). This design avoids the dependency loop between secure routing and online certificate distribution, and eliminates the need for a separate out-of-band certificate distribution mechanism. We strive to make the in-band distribution protocol efficient and capable of supporting complex operations such as certificate revocations and key rollovers (§ 4).

Third, we design IPA to be compliant with the existing protocols to be gradually adoptable. It uses the BGP optional and transitive attributes to carry IPA-specific information so that legacy ASes can pass this information to deployed ASes without interpreting them (§ 7.3.1). Different ASes can deploy IPA at different times without a "flag day." Furthermore, because we use the top-level reverse DNSSEC hierarchy to bind IP prefixes to their owners' public keys, the ASes who obtain their IP prefixes from the Internet registries can obtain their prefix ownership certificates from the registries without depending on other infrastructures. This feature enables those ASes, which amount to 78% of all ASes on today's Internet (§ 7.3.2), to form a deployed "club" to prevent various network-layer attacks within the club (§ 5).

We further show how IPA enables several security building blocks, including a secure routing protocol such as S-BGP [38], a source authentication system [43], and a DoS defense system [45] (§ 5). These security building blocks are also gradually adoptable [26, 43, 45], and together can prevent prefix hijacking, route forgery, and source address spoofing attacks, and suppress DoS flooding traffic near its sources.

We have implemented IPA using XORP [33] and integrated other security modules with it (§ 6). We evaluate IPA's performance and adoptability using trace-driven experiments (§ 7.2), live Internet experiments (§ 7.3.1), and analysis (§ 7.3.2). The results suggest that IPA is lightweight and gradually deployable in the current Internet. Our trace-driven experiments show that IPA's query overhead on an Internet registry's DNS servers is less than 0.1% of a single root DNS server's regular workload. Its in-band certificate distribution protocol introduces modest overhead to a router. A single-threaded IPA implementation running on a commodity PC can process all messages a RouteViews server [53] receives at their arrival rate. We expect that the server's workload is representative of a large ISP's BGP router's workload, because the number of peers it has (37) is the top 6% largest among all ASes [8].

Our live Internet experiments show that IPA's protocol messages piggybacked in BGP can pass standard-compliant legacy routers. Our analysis suggests that IPA lowers the deployment cost for early adopters compared to previous work that requires dedicated PKIs [38, 46, 56, 57], but offers equivalent or stronger security strength. Thus, it is more likely to be adopted.

To the best of our knowledge, IPA is the first design that brings accountability to the Internet in a secure, lightweight, and gradually adoptable manner.

2 System Models and Goals

Before we present the IPA design, we first describe its system models and design goals.

2.1 System Models

Network Model: IPA adopts the same two-level hierarchical network model (nodes and ASes) as the present Internet. For inter-AS routing and forwarding, we treat an AS as one trust and fate-sharing unit. AS boundaries are also trust boundaries. For clarity, we abstract each AS as a node when describing AS-level operations.

Trust Model: IPA assumes the same external trust entities as the present Internet. The global root of trust is the Internet Assigned Numbers Authority (IANA).

Threat Model: We assume that both hosts and routers can be compromised. Compromised nodes (hosts or routers) can collude into groups and launch arbitrary attacks. We also assume that an AS may be malicious, and malicious ASes can also collude.

2.2 Design Goals

IPA's central design goal is to securely bootstrap accountability in the Internet with lightweight and adoptable enhancements. We elaborate it in more detail.

Secure: IPA aims to enable cryptographically provable network-layer identities. As we show in § 5, this ability further enables various security modules that can prevent prefix hijacking [34, 38], route forgery [34, 38], source address spoofing [43], and DoS flooding attacks [45].

Lightweight: We aim to introduce only lightweight enhancements to the Internet. We believe that enhancing the existing infrastructures with new functions has lower deployment costs than rolling out new global infrastructures. For this reason, IPA does not require new global infrastructures, unlike [12, 38, 57]; nor does it require trusted hardware at end systems (although it can help), unlike [20]. Moreover, we aim to add little performance overhead to the deployed Internet base.

Adoptable: We aim to make IPA adoptable, which implies two sub-goals:

- **Gradually Deployable:** We aim to make IPA compatible with the legacy Internet and ready to be deployed on the Internet. IPA-enabled ASes (or hosts) should be able to run IPA-related protocols even if they are connected by legacy ASes.

- **Incentivizing Early Adoption:** IPA should require low deployment costs and provide immediate security benefits to early adopters to incentivize deployment. That is, the group of early adopting ASes should gain security benefits within the deployed region without requiring other entities outside the group to deploy IPA.

3 Overview

This section presents a high-level overview of IPA. We present more design details in the following section. IPA uses two key mechanisms to be lightweight and gradually deployable: 1) it uses the top-level reverse DNSSEC infrastructure as a lightweight PKI to bind an IP prefix to its owner's public key; and 2) it uses the BGP routing system to distribute IP prefix certificates in-band.

3.1 A Hybrid Approach to Secure Identifiers

The present Internet uses two types of identifiers: 1) a hierarchically allocated IP address (or prefix) to loosely identify a network attachment point (or a group of them in the same network), and 2) a flat AS number to identify an autonomous system. IANA is the root of trust and the owner of all IP addresses, *i.e.*, the owner of `0/0`. It delegates sub-prefixes to RIRs, which in turn delegate even smaller sub-prefixes to ASes. ASes may further subdelegate IP prefixes to their customers. Figure 1 shows an example of the address delegation hierarchy.

To be gradually deployable, IPA retains the hierarchical structure of IP addresses, and uses the existing chain of trust in the IP address allocation process to bind an IP prefix to its owner's public key. Since ASNs do not have a hierarchical structure, IPA replaces them with ASes' self-certifying identifiers, *i.e.*, the hash of their public keys. This design reduces the deployment overhead at an Internet registry, as a registry need not bind an AS's identifier to its public key. This new ASN format can be gradually deployed in a manner similar to how the 32-bit ASN was recently deployed [55].

3.2 DNSSEC as a Lightweight PKI

The IPA design uses the top-level DNSSEC infrastructure as a lightweight PKI for Internet registries to issue IP prefix delegation certificates. DNSSEC is originally designed to protect the integrity of DNS replies. Similar to a PKI, it allows a parent entity to use its key to certify a DNS zone delegation to a child entity. Each zone owner signs the DNS records in its zone, and publishes their signatures in DNS for verification. When a client performs a DNSSEC query for a domain name, it can verify the authenticity of the answer by following the DNS hierarchy to obtain the relevant DNSSEC records.

Using DNSSEC to certify IP prefix delegation has several advantages. First, we can create a one-to-one mapping between a reverse DNS zone delegation and an IP prefix delegation, as the reverse DNS hierarchy and the IP address hierarchy share the same root (IANA). For example, when IANA delegates an IP prefix `165/8` to an RIR (ARIN), it can also delegate the corresponding reverse DNS zone, `165.in-addr.arpa`, to ARIN (Figure 1). This delegation further enables ARIN to create a one-to-one mapping between the IP sub-prefixes and the reverse DNS zone's sub-delegations, *e.g.*, delegating `165.72/16` and `72.165.in-addr.arpa` to an AS (AT&T). A prefix owner can use the DNSSEC records that certify its reverse DNS zone delegation as a certificate authorizing its prefix ownership (§ 4.1). We refer to this type of certificate as an *IP prefix delegation certificate* or a *prefix certificate*. This design reduces IPA's deployment costs at an Internet registry, as it need not maintain a separate PKI to certify IP prefix delegations.

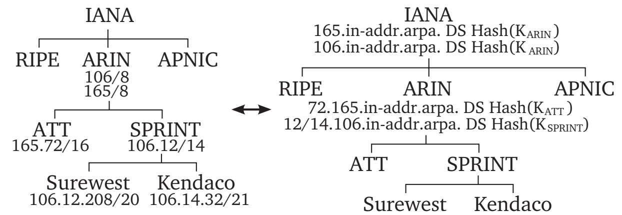


Figure 1: ***Left*: the IP prefix allocation hierarchy; *Right*: the corresponding DNSSEC records that bind the prefixes to their owners' public keys.**

The second advantage is that Internet registries are rapidly deploying DNSSEC [7, 29, 50]. The root zone was signed in July 2010 [19], and later the `arpa` and the `in-addr.arpa` zones. IANA will further sign the sub-zone delegations from `in-addr.arpa` in late March 2011 [7]. Moreover, the three largest RIRs, ARIN, RIPE, and APNIC, have all stated in their websites that they are ready to or will soon be ready to sign reverse zone sub-delegations [1, 2, 5]. Since these RIRs own 142 out of 175 sub-zones of `in-addr.arpa` [14], we expect that the top-level reverse DNSSEC will soon be fully deployed by all Internet registries.

Finally, because DNSSEC supports online queries, an Internet registry can use it to publish new IP prefix certificates to support key rollovers (§ 4.5) or revocations (§ 4.2), in addition to issuing certificates. An AS can query the DNS to download its up-to-date prefix certificates and the Internet registries' revocation lists.

3.2.1 IP Prefix Sub-delegation

After an AS obtains its IP prefixes, it may delegate subprefixes to its customers. For instance, Sprint in Figure 1 allocates a sub-prefix `106.12.208/20` to its customer Surewest. The IPA design allows an AS to flexibly choose the infrastructure it uses to manage these sub-delegation certificates. An AS can choose to use DNSSEC, as does an Internet registry. Alternatively, it

may use a certificate authority server to issue the IP prefix certificates. In the latter case, an AS should also support a certificate publishing mechanism (*e.g.*, a secure web server or an FTP server) to enable its customers to download their up-to-date certificates online. This requirement is to support automatic key rollovers (§ 4.5). We believe that an AS has incentives to manage and publish its customers' certificates, because this effort can protect its customers from prefix hijacking attacks.

For clarity, in the IP prefix delegation process, we refer to the delegator as the *parent owner*, and the delegatee as the *child owner*.

3.3 In-band Certificate Distribution

To prevent routing attacks, ASes must use a secure routing protocol (*e.g.*, S-BGP [38], § 5.1) to validate prefix origins and AS paths in BGP messages. This requires ASes to first obtain valid IP prefix certificates.

IPA uses BGP itself to distribute these certificates in-band to ASes that need them. That is, when an AS originates an IP prefix in a BGP message, it piggybacks the chain of certificates that can prove its prefix ownership in the message. We use a BGP feature, the transitive and optional path attribute, to carry the certificates. An AS can first obtain the chain of certificates offline when it obtains the IP prefix from its parent AS or an Internet registry. Later, it can periodically download the full chain of the latest certificates, as we will describe in § 4.5.

This design has several advantages. First, it avoids the dependency loop between secure routing and online certificate distribution. If we use an alternative approach where each AS downloads the prefix certificates from online distribution servers (*e.g.*, DNSSEC servers), a dependency loop between routing and certificate distribution may occur. This is because to obtain a prefix p's certificate C_p, an AS X must first establish a valid path to an AS Y that hosts C_p's distribution server. Recursively, to establish a valid path to AS Y, X must validate the BGP messages advertising AS Y's prefixes, which requires AS X to have obtained AS Y's prefix certificates. These certificates may be served by a distribution server in yet another AS Z, and to establish a valid path to Z, X needs the certificates for Z's prefixes, and so on. These dependencies may eventually form a loop, preventing AS X from obtaining the certificates needed to validate the prefix p's ownership.

In contrast, in-band distribution does not introduce such dependencies. This is because it does not require an AS to establish an *a priori* valid path to an online distribution server. BGP messages are propagated hop-by-hop (at the AS level). An AS will first obtain valid certificates from its neighbors, and then from its neighbors' neighbors, and so on, until it obtains the valid certificates from all ASes in the routing system.

Second, in-band distribution lowers deployment costs, as it does not need an out-of-band channel to distribute the certificates, unlike [38, 56]. IPA also uses standard BGP features to encode the certificates so that different ASes may gradually adopt the distribution mechanism without breaking BGP.

Finally, including a prefix p's full chain of certificates ensures that any AS that receives a BGP message originating p can immediately validate p's owner's public key. This further ensures that an AS can promptly validate the prefix origin and AS path in the BGP message (§ 5.1) and propagate the message and the chain of certificates further to its neighbors. These neighbors can in turn use the certificates to validate the BGP message and propagate it further, until all ASes have received and validated the BGP message. We refer to this property as *liveness*, and provide a formal proof of it in [42]. We discuss how to validate a certificate in § 4.4.

Attaching a full chain of certificates in a BGP message incurs significant communication overhead. IPA uses a simple but effective technique to reduce this overhead: each AS caches the certificates that it has sent to a neighbor and only sends to the neighbor the certificates that it has not sent yet. We describe it in more detail in § 4.3.

4 Design Details

This section presents more design details of IPA, including how to use DNSSEC records to encode an IP prefix certificate (§ 4.1), certificate revocation (§ 4.2), efficient certificate distribution (§ 4.3), certificate validation (§ 4.4), and key management (§ 4.5).

4.1 DNSSEC Records as IP Prefix Certificates

IPA uses three types of a reverse DNS name's resource records to encode a prefix certificate: the designated signer (DS) record, the public key (DNSKEY) record, and the signature (RRSIG) record of the DS record.

Figure 2 shows the DNSSEC records that form the certificate for the prefix `165/8`, which IANA allocates to ARIN (Figure 1). These records are associated with the DNSSEC entry `165.in-addr.arpa` created by IANA. IANA uses the DS record to store the hash of ARIN's pubic key, and signs the DS record using its private key. It sets the inception and expiration times of the signature record (RRSIG) to the inception and expiration times of the prefix allocation, and publishes the entry `165.in-addr.arpa` on its DNS servers. This process follows the standard DNSSEC practice, and also applies to IPv6 address allocation.

A slight complication arises as not all IP address allocations fall on a reverse DNS domain boundary. For instance, as shown in Figure 1, ARIN may allocate an IP prefix `106.12/14` to Sprint. We address this issue by extending the encoding format of a re-

165.in-addr.arpa DNSKEY KARIN (290 bytes)
165.in-addr.arpa DS Hash(KARIN) (50 bytes)
165.in-addr.arpa RRSIG DS (312 bytes)

Figure 2: **This figure shows the DNSSEC records that encode the prefix `165/8`'s certificate. The size of each record is estimated assuming that the signatures are generated using 2048bit RSA/SHA-1.**

verse DNS name. For instance, we use the reverse DNS name `12/14.106.in-addr.arpa` to encode the IP prefix `106.12/14`. The encoding/decoding rules are straightforward and compatible with the DNS standard [47]. We omit them due to the lack of space, but describe them in [42]. We choose not to use the existing techniques that support classless reverse zone delegations [27, 30], because they either only support allocations in chunks smaller than a `/24` prefix [30], or are no longer supported by popular DNS servers [9, 27].

4.2 Revoking an IP Prefix Certificate

An Internet registry or an AS may revoke a certificate allocated to a child before it expires. This may occur if the prefix is re-assigned to a new child owner, or the child owner's key is compromised, or the child owner violates the terms of use or switches to a different ISP.

In the IPA design, a parent owner issues a new prefix certificate to explicitly revoke the old one. The new certificate binds the IP prefix to a new public key with a newer inception time. The new key could be a new child owner's key, or the present child owner's new key, or the parent's own key if it reclaims the IP prefix from a child.

As we discuss in § 3.3, IPA distributes IP prefix certificates in the routing system for ASes to validate routing messages. To use a certificate to validate a routing message, an AS must know whether the certificate has been revoked or not. IPA uses both *push* and *pull* mechanisms to notify an AS of a certificate's revocation status.

Pushing New Certificates via Routing: Because a new certificate explicitly revokes an old one, a new certificate's owner can immediately announce the new certificate in BGP using the in-band distribution mechanism to notify other ASes of the old certificate's revocation.

Periodic Pulling From Internet Registries: When an Internet registry revokes a prefix certificate, the registry may be unable to notify other ASes using the push-based mechanism, because it does not participate in routing. We use a DNSSEC-based revocation list to address this problem. A revocation list includes the set of IP prefixes an Internet registry reclaims from its children, or re-assigns to its children that are also Internet registries. The registry can publish the list using a TXT record with a special DNS name, *e.g.*, `revoked.arin.in-addr.arpa`, and sign the list using DNSSEC. An entry in a revocation list includes the revoked IP prefix and the revocation time. It revokes any older prefix certificate signed by the same registry and whose address range overlaps with the revoked prefix.

Each AS periodically (*e.g.*, daily) downloads the revocation lists from all Internet registries to invalidate revoked certificates (§ 4.4). An AS does not query DNS at the certificate validation time to reduce DNS load. Periodic downloads may delay a certificate's revocation, but we consider this delay acceptable, as it will not lead to prefix hijacking attacks. Only the IP prefixes not allocated to any AS will suffer this delay, as an AS that owns an IP prefix can immediately announce its new certificate in BGP to revoke the old one.

4.3 Efficient Certificate Distribution

As we describe in § 3.3, IPA uses a BGP message itself to distribute the full chain of certificates of the IP prefix that the message advertises. We now describe how to make this in-band distribution protocol efficient.

Each AS maintains several certificate caches to record what it has sent to a neighbor and to maintain certificate validation state, as shown in Figure 3. The caches include: 1) an incoming certificate cache that stores all certificates received from its neighbors; 2) a trusted certificate cache that stores the certificates it has validated; and 3) a per-neighbor outgoing certificate cache that records the hash of each certificate it has sent to the neighbor. An AS organizes the certificates in its trusted cache in a tree-like structure following the IP allocation hierarchy to assist certificate validation (§ 4.4).

When an AS receives a prefix certificate from a neighbor, it first stores the certificate in its incoming cache, and then validates the certificate as we describe next. When the AS sends a BGP message to a neighbor announcing the IP prefix, it will retrieve the full chain of certificates from its trusted certificate cache, and compare them with those in the neighbor's outgoing certificate cache. It will only send the certificates that are not in the neighbor's outgoing cache, and then insert them in the outgoing cache to avoid sending them to the neighbor again.

When an AS loses the peering connection to a neighbor, *e.g.*, due to a router reboot or link failure, it will remove all entries in the neighbor's outgoing cache. When the AS resumes its connection with the neighbor, it will re-send the full chain of certificates for each prefix it announces to the neighbor.

4.4 Validating IP Prefix Certificates

When an AS receives a BGP message that advertises a prefix p_n and includes a list of certificates from a neighbor, it must validate these certificates to verify p_n's

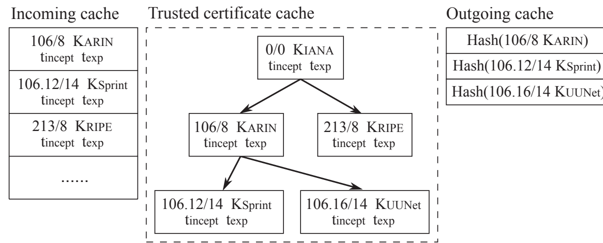


Figure 3: **An example of the certificate caches an AS maintains. It shows only one outgoing cache of the AS.**

owner's public key. It considers a prefix p_n's certificate C_{p_n} valid if C_{p_n} meets the following conditions:

1. C_{p_n} is not on any Internet registry's revocation list or revoked by a newer certificate (§ 4.2).
2. C_{p_n} has a valid parent certificate $C_{p_{n-1}}$ such that 1) C_{p_n} is signed by its parent certificate $C_{p_{n-1}}$'s private key; 2) p_n is a subset of its parent certificate's prefix p_{n-1}. If p_n is the prefix `0/0`, C_{p_n} need not have a parent but must be self-signed by IANA.

Algorithm 1 shows the pseudo-code for the validation algorithm. Most steps of the algorithm check whether C_{p_n} satisfies the above conditions. We note two things. First, if C_{p_n} does not have a valid parent certificate, C_{p_n} becomes unverifiable. Unverifiable certificates may exist temporarily during a key rollover event (§ 4.5). The algorithm returns failure but leaves C_{p_n} in the incoming cache, as it may become valid later after its parent certificate has arrived. Second, the last section of the code (line 23–26) adds the newly validated C_{p_n} to the AS's trusted cache and checks whether any previously unverified certificate C_i is now verifiable, which may happen if C_{p_n} is its parent. If such a certificate C_i exists, the algorithm recursively validates it and its child certificates.

Algorithm 1 validate(C_{p_n}): pseudo-code to validate the certificate C_{p_n} in an incoming BGP update message msg.

Input: C_{p_n}, the incoming certificate to be validated; p_n, the prefix of C_{p_n}; msg, the incoming BGP message; $cache_{tr}/cache_{in}$, the current trusted/incoming certificate cache; $rlist[r]$, the most recent revocation list of registry r

```
1:  if is_registry(C_{p_n}.signer)
    and p_n ∈ rlist[C_{p_n}.signer] then
2:    cache_{in}.remove(C_{p_n})
3:    return false
4:  end if
5:  C_{p_{n-1}} ⇐ cache_{tr}.lookup_parent(C_{p_n})
6:  if C_{p_{n-1}} == NULL then
7:    C_{p_{n-1}} ⇐ msg.lookup_parent(C_{p_n})
8:    if C_{p_{n-1}} == NULL or not validate(C_{p_{n-1}}) then
9:      return false
10:   end if
11: end if
12: for C_s ∈ cache_{tr}.get_children_certs(C_{p_{n-1}}) do
13:   if overlap(p_n, p_s) then
14:     if C_{p_n}.inception > C_s.inception then
15:       cache_{tr}.recursive_remove(C_s)
          // remove all certificates in C_s's subtree
16:       cache_{in}.remove(C_s)
17:     else
18:       cache_{in}.remove(C_{p_n})
19:       return false
20:     end if
21:   end if
22: end for
23: cache_{tr}.insert(C_{p_n})
24: for C_i ∈ cache_{in} and C_i ∉ cache_{tr} do
25:   validate(C_i)
26: end for
27: return true
```

4.5 Key Management

Like any cryptography-based system, IPA's accountability builds on the secrecy of private keys. In addition to the standard practice to protect secret keys, IPA takes two additional measures: 1) separating an AS's identity keys from the keys the AS uses to sign routing messages, and 2) periodic key rollovers.

4.5.1 Separating Identity Keys from Routing Keys

To secure routing, an AS must store its private key online to sign routing messages (§ 5.1). Yet it is desirable to keep a private key offline to reduce the risk of key compromise. To balance security and functionality, IPA separates an AS's identity keys from the keys it uses to sign routing messages. We refer to the pair of keys associated with an AS's self-certifying identifier as its identity keys, or its identity key when we refer to either the AS's private or public key.

An AS generates a separate pair of public/private keys to sign routing messages. We refer to this pair of keys as an AS's routing keys. For each IP prefix it owns, an AS will use its identity key to sign a routing certificate that binds the IP prefix to its routing key. The AS keeps its identity private key offline, and uses its routing private key to sign routing messages. An AS will include a prefix's routing certificate in its BGP messages. Other ASes can validate it using the algorithm described in § 4.4.

4.5.2 Routing Key Rollover

By separating identity keys from routing keys, an AS can periodically expire its routing keys, issue new ones, and sign its new routing certificates with its identity key, all without changing its identifier, or re-signing its prefix sub-delegation certificates.

4.5.3 Identity Key Rollover

An entity should also change its identity keys periodically to improve security. To change its identity keys, an entity must 1) request new certificates from its parents, 2) revoke its old certificates, and 3) re-sign each child certificate with its new private key. As can be seen, this process is more complicated than routing key rollover. Thus, an entity should change its identity keys at a lower frequency than its routing keys.

A key challenge we face is how to make a child certificate remain valid throughout a parent key rollover event so that other ASes can verify the child's routing messages. We address this challenge by "pre-releasing" a child's new prefix certificate, a technique similar to how DNSSEC manages key rollovers [39]. With this mechanism, both a child's old and new certificates remain valid during a key rollover event.

For clarity, we first describe the identity key rollover process for an AS, and then for an Internet registry. Figure 4 shows this process. Let D be an AS that wishes to rollover to a new identity key K_{new}. D will first use its old key K_{old} to generate a *transient* certificate certifying K_{new} for each prefix it owns. The transient certificates are only available during key rollovers, and will expire afterwards. Meanwhile, D generates a new certificate for each sub-prefix it delegates to a child using its new key K_{new}. D will also generate new certificates to certify its routing keys using K_{new}. At this point, both K_{old} and K_{new} are valid identity keys of D, because each of them can be certified by a valid chain of certificates, as shown in Figure 4(b). D will then publish the child certificates signed using its new key K_{new} via its certificate publishing system as described in § 3.2.1.

Each AS will periodically (*e.g.*, once a day) query its certificate issuers' publishing systems to download its latest chains of certificates. If the AS obtains IP prefix allocations directly from an Internet registry, it will query the corresponding reverse DNS names of its IP prefixes starting from the root servers. Otherwise, the AS queries its parent ASes' certificate publishing systems. This online certificate downloading step does not have a dependency loop with routing, because each AS's old certificate chain is already in the routing system, and can be used to establish valid paths. If an AS C downloads a new certificate signed by its parent D's new key, it will immediately announce its new certificate in BGP. Other ASes will consider C's new prefix certificate valid, because it is certified by a valid chain of trust, including the link provided by the parent D's self-signed transient certificate, as shown in Figure 4(b).

Finally, the rekeying AS D requests each of its parents P that has delegated an IP prefix to its old key K_{old} to issue a new certificate to its new key K_{new}, after waiting for a long enough period d. The waiting period d should be long enough to ensure that each child AS of D has successfully downloaded and announced its new certificates in BGP. D can then announce its new certificate for its new key K_{new} in BGP to revoke its old certificate. The child AS C's certificate will remain valid, as shown in Figure 4(c). An AS D will also re-send its BGP routes to its neighbors using its new identifier.

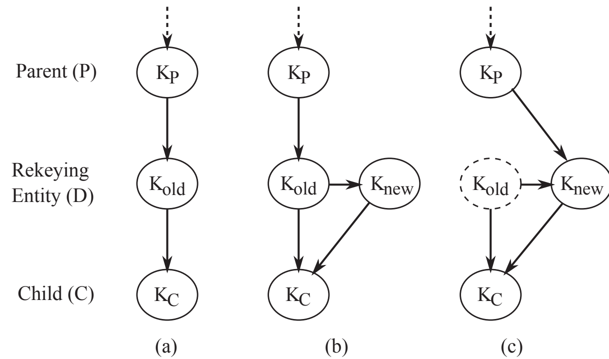


Figure 4: **This figure shows IPA's key rollover process. Each node represents a key; an arrow points from a parent's signing key to a child's signed key. Figure (a) shows the chain of trust before the key rollover; (b) shows the chains of trust during the key rollover, where the rekeying entity D signs a transient certificate to certify its new key K_{new} using its old key K_{old}; (c) shows when the key rollover process finishes, the old key K_{old} becomes invalid.**

An Internet registry's key rollover procedure is similar, except that the registry need not announce a new certificate in BGP, as its children will obtain it via DNSSEC.

4.5.4 Recovering From Key Compromise

With the preventive measures we describe above, we expect key compromise to be a rare event in IPA. For completeness, we briefly describe how to recover from it and leave the details to [42].

Recovering from key compromise resembles a key rollover event, except that an entity may resort to contacting its parents and children offline to obtain its new certificates and distribute its children's new certificates. This is because when an attacker compromises an entity's identity keys, it may also hijack the entity's IP prefixes, making it unreachable online.

5 Use of IPA

In this section, we describe how IPA enables various security modules that collectively achieve accountable routing and forwarding, and DoS attack mitigation. Each of the modules we describe here is also gradually adoptable [26, 38, 43, 45].

5.1 Accountable Routing

IPA enables secure routing protocols such as S-BGP [38], because it provides ASes with the necessary

certificates to achieve origin authentication and AS path authentication.

Origin Authentication: An AS O that owns a prefix p can now sign its BGP messages when it announces the prefix, because other ASes can use the chain of certificates piggybacked in the BGP messages to verify the secure binding between the prefix p and O's public key (§ 3.3), preventing other ASes from originating p.

AS Path Authentication: Each transit AS can sign a BGP update using its private key when it prepends its self-certifying AS identifier to the update and propagates the update to a neighbor. A malicious AS cannot forge another AS's identifier, nor can it truncate the AS path, because it cannot generate a valid signature of another AS. A transit AS can piggyback its public key in a BGP message similar to how IPA distributes prefix certificates (§ 3.3). We can also apply the same caching technique described in § 4.3 to reduce the message overhead.

Self-certifying ASNs prevent path forgery, but raise a different security concern: an AS may mint arbitrary identifiers, which complicates BGP policy configurations. The IPA design addresses this concern by binding a self-certifying ASN to an IP prefix. If an AS path contains an ASN that is not a hash of a public key found in a valid IP prefix certificate, other ASes can consider the path not trustworthy, and configure their BGP policies to avoid this path. Moreover, an AS can use IP prefixes to configure its BGP policies, because other ASes cannot arbitrarily change their IP prefixes.

5.2 Accountable Forwarding

The ability to securely sign BGP messages enables Passport [43], a system that can achieve both packet source authentication and forwarding path inconsistency detection. Passport uses a distributed Diffie-Hellman key exchange piggybacked in BGP to establish a shared secret between every pair of ASes. With IPA, an AS O can sign the BGP messages that originate both its prefixes and its Diffie-Hellman public value. Other ASes can securely bind the secrets they share with AS O with O's prefixes to enable AS-level packet source authentication and path inconsistency detection.

Packet Source Authentication: To authenticate a packet's source address, a source AS stamps a sequence of message authentication codes (MACs) into a packet header using the secret keys it shares with each AS en route to the packet's destination. ASes along the path can re-compute the MACs to validate the packet's origin AS, as packets with spoofed source addresses will not have valid MACs.

Forwarding Path Inconsistency Detection: A malicious AS may attempt to advertise one legitimate AS path but forward packets along a different one that conflicts with a source AS's routing policies. The MACs that a source AS stamps into a packet header can help detect this misbehavior. This is because if a packet's forwarding path differs from the AS path its source AS selects to use, an AS on the path will detect an invalid MAC, but the destination AS will detect a valid one. A destination AS can use this discrepancy to notify the source AS of the forwarding path inconsistency.

5.3 DoS Attack Mitigation

Finally, because IPA enables source authentication, it also enables DoS defense systems that use authentic source addresses to suppress attack traffic near its sources, *e.g.*, a filter based system StopIt [44], or NetFence [45], a system based on unspoofable congestion policing feedback.

As an example, we describe briefly how NetFence can use IPA to suppress DoS flooding traffic near its sources. NetFence introduces a secure congestion policing framework in the network. A NetFence packet carries unspoofable congestion policing feedback in a shim layer. An on-path AS updates this feedback to notify an access router of its local congestion conditions, and an access router uses this feedback to regulate a sender's sending rate. The on-path AS and the source AS use the secret they share via Passport to protect this feedback from being tampered by malicious routers or end systems. When malicious sources and receivers collude to flood a link in the network, NetFence provides a legitimate sender its fair share of bandwidth. When a receiver is an innocent DoS victim, NetFence enables the receiver to use the unspoofable congestion feedback as network capabilities [59] to suppress the bulk of unwanted traffic.

We introduce AS-level hierarchical accountability to NetFence to accommodate IPA's self-certifying ASNs. The original NetFence design uses AS-level queues at a router to hold each source AS accountable for its traffic. With IPA, we use hierarchical queuing [24] that follows the IP allocation hierarchy to hold each AS accountable. That is, the traffic from all IP prefixes allocated to an AS's public key will share one queue; a router may subdivide the queue into multiple lower-level queues, if the AS delegates sub-prefixes to its customers, and so on. A router sets a queue's weight according to the size of the IP prefixes associated with the queue, not by the number of ASes sharing the IP prefixes. This mechanism prevents an AS from gaining unfair network resources by dividing its IP prefixes into many smaller ones and delegating them to minted identifiers.

6 Implementation

We have implemented a prototype of IPA's in-band certificate distribution mechanism (§ 3.3) using XORP [33]. The implementation includes a standalone C++ library

libipa that other BGP implementations can use. The library libipa implements certificate distribution and validation, and supports downloading revocation lists and new certificates from DNSSEC.

Our implementation addresses several practical issues that arise when an IPA router peers with a legacy router. First, we disable the optimization technique (§ 4.3) on an IPA router's interface facing a legacy router, because a legacy router does not cache any certificate or public key. Furthermore, legacy BGP has a 4KB limit on the size of an update message. To bypass this limitation, an IPA router breaks a message longer than 4KB into smaller ones, each of which carries a subset of the certificates and public keys of the original message. The router sends them in sequence to its legacy neighbor. The IPA router waits for a period of time longer than the BGP's MRAI timer (*e.g.*, a few minutes) between sending out two consecutive messages to prevent the first message from being overwritten by the second one.

We have also extended previous implementations of S-BGP, Passport, and NetFence and incorporated them into the IPA prototype. We defer a systematic evaluation on the integrated architecture to future work.

7 Evaluation

In this section, we evaluate IPA along four dimensions. First, we use small-scale testbed experiments to validate the design and implementation. Second, we use trace-driven benchmarks to measure the design's performance and overhead. Third, we use live Internet experiments and analysis to evaluate the design's adoptability. Finally, we analyze IPA's security properties.

7.1 Testbed Experiments

We use DETERlab [28] experiments to validate the design and implementation of IPA. These experiments include 1) bootstrapping experiments, 2) key rollover experiments, and 3) prefix hijacking experiments. We sample a small test topology from the AS-level Internet topology inferred from BGP table dumps. This topology includes six university ASes and all ASes on the shortest AS paths between the six ASes. It contains 17 ASes and 54 uni-directional links. We desire to run larger-scale experiments, but are limited by the number of testbed machines we can obtain. For simplicity, we assume each AS owns one prefix, and choose the prefix to be the largest one the AS owns in reality. Finally, we assume all ASes use DNSSEC to issue and publish their certificates, and use the signing tool included in BIND9 [3] to generate the certificates. The topology includes four levels of IP prefix allocation: IANA, RIRs, top-level ASes, and customer ASes. We randomly pick three ASes to host the root and two RIRs' DNSSEC servers. We assume each AS's DNSSEC server is inside its network. Each node

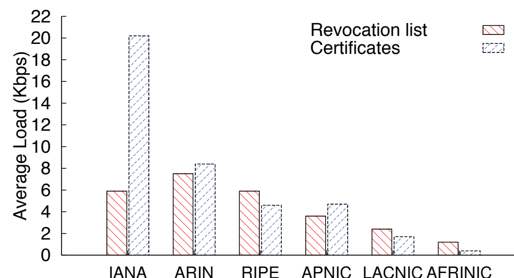


Figure 5: **This figure shows the average DNS traffic load of each Internet registry to serve the revocation list and the IP prefix certificates.**

in a testbed experiment corresponds to an AS. Each AS is configured with an initial IP prefix certificate chain.

We summarize the testbed experiment results as follows. In a bootstrapping experiment, each node can validate all certificates and store them in its trusted cache, suggesting that the system can successfully bootstrap, consistent with the liveness property of IPA's in-band certificate distribution protocol (§ 3.3). In a key rollover experiment, the rekeying ASes can successfully propagate their new certificates, and each prefix always has at least one valid chain of certificates during the rollover period. Finally, we run our S-BGP module using the certificates distributed by IPA. We launch a prefix hijacking attack from an AS. All other ASes reject the update message because there does not exist a certificate chain certifying the AS's ownership of the hijacked prefix.

7.2 Performance

IPA adds overhead to both DNS and BGP. We use trace-driven benchmarks to evaluate this overhead. The results show that IPA's overhead on DNS and BGP is acceptable. We use a PC with Xeon 3GHz CPU and 2GB memory to run all of our experiments unless otherwise noted.

7.2.1 DNS Overhead

IPA uses a signed TXT record in DNS to publish an Internet registry's revocation list (§ 4.2). An AS periodically downloads the revocation list from each registry. Each entry in a revocation list can be encoded in $\leq$30 bytes ($\leq$18 bytes for an IPv4 prefix in the dotted-decimal format, one byte for space, 10 bytes for the revocation time, and one byte for the line break). A publisher can compress a list (*e.g.*, using gzip) to reduce overhead. An AS also needs to download the list's signature ($\sim$300 bytes) and a few other DNSSEC records.

We assume that at any time, a registry at most revokes 1% of the total prefixes that it owns and does not re-allocate them to others. We use gzip to compress each revocation list, and use base64 to encode a compressed list so that it can be stored as a text record. The BGP report of February 2011 [15] shows that there are a total of

BGP Table Dump	
Date collected	08/01/2010
Number of ASes	35728
Number of IP prefixes	337K
BGP Update Trace	
Vantage point	route-view2.oregon-ix.net
Number of peers	37
Date collected	08/01/2010∼08/31/2010
Number of updates	118 million
Average arrival rate	44.1 updates/s

Table 1: **This table summarizes the BGP data we use in evaluating IPA's routing overhead.**

37K ASes on the Internet. We assume that an AS downloads a revocation list once per day. This downloading frequency is acceptable, because it at most allows a prefix's previous owner to use the prefix for one extra day.

Figure 5 shows the average traffic load for serving the list at each Internet registry's DNS servers. As can be seen, even for the busiest registry ARIN, the estimated communication overhead is less than 10Kbps. This overhead is negligible compared to the regular load of a top-level DNS server, *e.g.*, the "M" root DNS server's regular load is over 32Mbps [10].

In the IPA design, an AS may also periodically download its certificate chains from the Internet registries to deal with key rollovers (§ 4.5). To evaluate this overhead, we assume that all ASes publish the IP prefix certificates they delegate to their children using DNSSEC. This places an upper bound on the top-level DNS servers' load. Each certificate includes three DNSSEC records and is about 650 bytes long (§ 4.1). We assume that each AS downloads its certificates once every day for each prefix it owns. Figure 5 shows the average traffic load from all registries for serving the certificate downloads. As can be seen, the IANA's DNS servers have the highest certificate serving overhead, but it is still much lower than a root DNS server's regular load, which suggests that IPA is unlikely to stress DNS.

7.2.2 Routing Overhead

We use trace-driven experiments to evaluate the overhead of IPA's in-band certificate distribution mechanism. We obtain a real BGP update trace from a RouteViews server [53]. Table 1 summarizes the BGP data we use. We then add IPA specific fields and updates to the trace to obtain a synthetic IPA BGP trace. We use the synthetic IPA trace to estimate the message overhead of distributing IP prefix certificates in-band. We also feed the IPA trace to a PC router running our IPA implementation, and measure the router's processing and memory overhead.

We generate the IPA BGP trace in three steps: 1) inferring IP prefix delegation hierarchy; 2) adding certificates for newly allocated and re-assigned prefixes; and 3) adding updates triggered by key rollover events. We

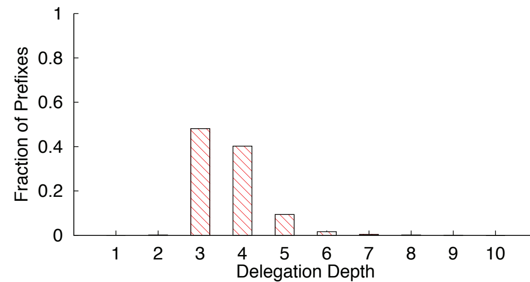


Figure 6: **The distribution of the depth of each prefix in the inferred IP prefix delegation hierarchy.**

describe each step in more detail.

First, we infer a prefix's delegation hierarchy to decide what certificates to add to a BGP update message announcing that prefix. We use a BGP table dump to infer this information. If an AS originates an IP prefix in the BGP table, we assume that it is the prefix's owner. If a prefix p' includes another prefix p, and both prefixes appear in the BGP table, we infer that p''s owner AS delegates the prefix p to p's owner. We also combine the IP prefix allocation records obtained from RIRs and IANA's websites to build the entire IP prefix delegation hierarchy. Figure 6 shows the distribution of the depth of the inferred hierarchy. More than 80% prefixes have a delegation depth of 3 or 4, suggesting that most ASes obtain IP prefixes directly from the RIRs or from provider ASes that directly obtain IP prefixes from the RIRs.

Second, we add prefix certificates to BGP updates that announce newly allocated or re-assigned IP prefixes. According to the IPA design (§ 3.3), an AS only sends an IP prefix certificate to a neighbor if it has not sent the certificate to the neighbor before. Thus, after the routing system has bootstrapped, only two types of updates carry IP prefix certificates: 1) an update that announces a newly allocated or re-assigned prefix, and 2) an update that carries new certificates generated during key rollovers (§ 4.5) for a previously announced prefix. We treat any IP prefix that has not appeared in the trace before as a newly allocated prefix, and any prefix whose origin AS has changed as a re-assigned prefix. To estimate the upper bound on the message overhead, we add the full certificate chain to each BGP update announcing a newly allocated or re-assigned prefix.

Finally, we add the update messages triggered by key rollover events to the IPA trace. Let a key rollover interval be T_r seconds. We let each AS randomly choose a key rollover time t during the T_r interval. We then add BGP updates that include the rekeying AS's new certificates for all its prefixes and its child ASes' prefixes at time t in our trace. We add updates for both routing and identity key rollovers (§ 4.5). We assume that as an upper bound, each AS changes its routing keys once a week,

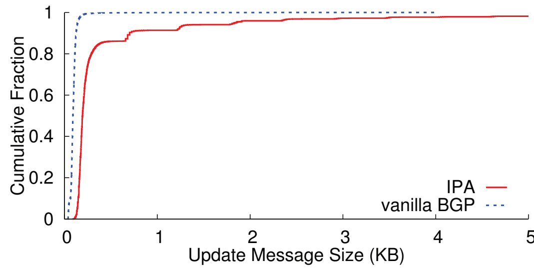


Figure 7: **The cumulative distribution of an IPA BGP update message size.**

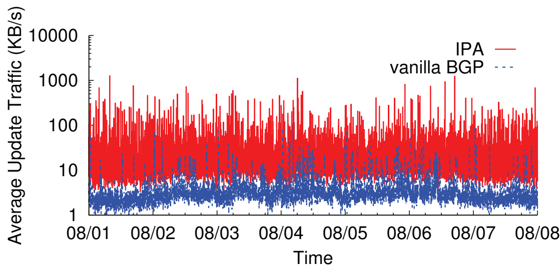


Figure 8: **The update traffic rate a RouteViews server sees averaged over 1-minute intervals during one week.**

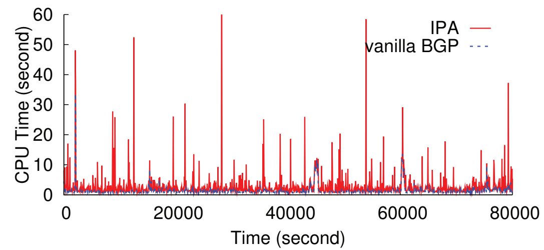


Figure 9: **The CPU time taken to process the messages received per 1-minute bin.**

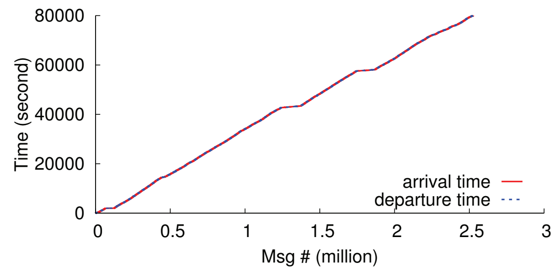


Figure 10: **The arrival and departure time of each message received during a day. The message number is in the unit of million (M).**

and its identity keys once a month.

Message Overhead: Figure 7 shows the cumulative distribution of an IPA message size in one day's trace (August 1, 2010). The distributions in other days are similar and hence omitted. For comparison, we also show the distribution of an original BGP message size. As can be seen, over 80% of the IPA messages are smaller than 500 bytes. Given that each IP prefix certificate is around 650 bytes (§ 4.1), we can infer that over 80% of the messages do not carry any certificate, indicating that the caching mechanism described in § 4.3 is effective in reducing message overhead.

Figure 8 shows the IPA BGP update rate averaged over 1-minute bins in one week (August 1–7). The results during other weeks are similar and are omitted for clarity. For comparison, we also show the vanilla BGP update rate. The RouteViews server we use peers with 37 large ISPs. So we expect that the update process it sees is representative of what a BGP router sees in a large ISP [8]. The rate shown in Figure 8 is the aggregate arrival rate over all peers of the server. As can be seen, IPA increases the update traffic rate compared to the vanilla BGP. The 1-minute average aggregate update rate is usually less than 200KB/s. Since there are 37 peers, each peer on average receives less than 6KB/s update traffic. We think this overhead is acceptable compared to today's core routers' link capacities (10Gbps or 40Gbps).

Processing Overhead: We evaluate an IPA router's processing overhead by measuring 1) the fraction of CPU time it takes to process IPA's BGP messages, and 2) each message's processing latency. We aggregate the BGP update messages into 1-minute bins to measure the CPU utilization. We feed the messages arrived in each bin to our IPA router implementation, measure the aggregate processing time, and compare it with the bin size.

Figure 9 shows the result during a one-day period (August 1, 2010) with 1-minute bins. The results for other days are similar and we omit them for clarity. For comparison, we also show the CPU time a XORP BGP router spends to process the original BGP trace. For each time bin, IPA takes more time to process the messages than the vanilla BGP, because it needs to validate new certificates piggybacked in the incoming messages. However, the CPU time that the router spends to process each 1-minute bin messages is usually less than 30 seconds, indicating that the router's CPU utilization is less than 50% and CPU is not a bottleneck. We may further improve our implementation's efficiency by applying instruction-level optimization to the RSA algorithm [40].

We further evaluate IPA's processing latency and examine whether it can keep up with the update arrival rate. We feed each update to the IPA router implementation according to the time it arrives. Figure 10 shows the arrival and departure time of each message. As can be seen, the arrival and departure lines almost overlap with each other, indicating that our implementation running on a commodity PC can keep up with the update arrival rate of the RouteViews server.

Memory Overhead: To evaluate IPA's memory over-

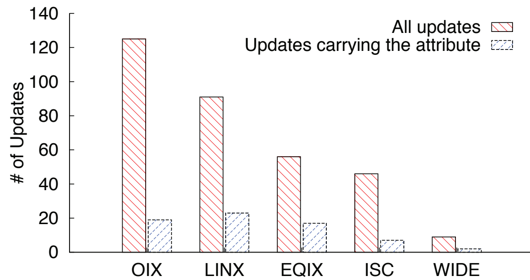


Figure 11: **The number of updates received by each RouteViews vantage point.**

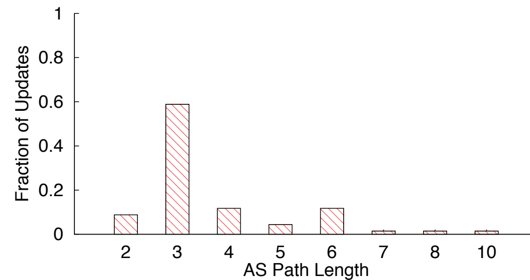


Figure 12: **The AS path length distribution of the received updates that carry the optional and transitive test attribute we inject. The path is from a RouteViews vantage point to the injection location.**

head, we feed the IPA BGP trace to our IPA implementation, and measure the memory needed to store all certificate caches. With our implementation, the trusted certificate cache consumes around 356MB memory using the BGP table data shown in Table 1. Our implementation stores only one physical copy for each certificate. The same certificates in different caches are pointers to the physical copy. The incoming cache uses ∼1.5MB memory to store the pointers. An outgoing cache uses at most 7MB, because it only need store a hash value for each certificate. This memory overhead is moderate because a router need not use these certificates in the packet forwarding time and can store them in low-cost DRAM.

7.3 Adoptability

In this section, we use real Internet experiments and analysis to evaluate IPA's adoptability. An adoptable design must satisfy two conditions: gradually deployable and providing incentives to early adopters.

7.3.1 Gradual Deployment

IPA uses the top-level DNSSEC infrastructure and BGP to certify and distribute IP prefix certificates. We evaluate whether early adopters can gradually deploy IPA in each system.

DNSSEC: First, we evaluate whether a legacy DNSSEC implementation can serve the DNSSEC records and revocation lists needed by IPA. We deploy a BIND9 DNS server which supports DNSSEC natively and has the largest installation base [16]. We use the DNSSEC signing tool bundled with the server software to generate the DNSSEC zone records for the IP prefixes allocated by IANA and all five regional Internet registries, and configure the server to serve the records and the revocation lists. We then use a legacy DNS client `dig` to fetch them. The `dig` client successfully retrieves all the records, indicating that the Internet registries can directly serve the DNSSEC records required by IPA without modifying DNS servers or breaking DNS clients.

BGP: We use BGP's transitive and optional path attributes to carry IPA-related fields. This design allows upgraded ASes to run the IPA protocols even if they are connected by legacy routers. This is because according to the BGP standard [51], legacy routers should forward any transitive and optional attribute.

To test IPA's compatibility with legacy BGP routers, we use a modified Quagga [11] BGP daemon to inject a BGP update with a transitive and optional attribute. We then monitor the propagation of this update from multiple RouteViews' vantage points. On August 27, 2010, we injected one such update to BGP using the BGP beacon platform maintained by RIPE RIS [13]. The update includes a previously unused prefix and a 3KB path attribute with an unknown type code 99. Figure 11 shows the number of updates observed by each RouteViews vantage point and among them how many still carry the attribute. For the updates still carrying the attribute, Figure 12 shows the AS path length distribution from their vantage points to the injection point. As can be seen, each vantage point observes at least one update carrying the attribute, and most of the updates carrying the attribute have successfully traversed multiple legacy ASes.

The RouteViews vantage points also receive many updates without the attribute. We suspect that this is caused by a Cisco software bug triggered by the injected update [17]. The bug causes certain Cisco router models to corrupt the path attribute. Consequently, a downstream router may reset the connection or remove the corrupted attribute. Given the prevalence of Cisco routers, we think that the result is encouraging. We expect that the affected routers will soon patch up this bug, and we will observe much more updates carrying the test attribute if we repeat this experiment.

7.3.2 Incentives for Early Adopters

We now discuss how the IPA design provides incentives for early adopters. Our analysis is based on the adoptability model presented in [26, 43]. The model assumes that each potential adopter is rational, and will have incentives to adopt a security mechanism if the security benefits outweigh the adoption costs. Because it is diffi-

cult to quantify costs, we use the model to qualitatively argue that IPA provides stronger incentives for adoption than previous work [34, 38, 46, 56, 57]. Thus, it is more likely to be adopted than previous work from a cost-effective perspective. We do not claim that IPA will be adopted, as many other factors (*e.g.*, politics) may affect the adoption process.

IPA's deployment involves four key parties: Internet registries, ASes, router vendors, and OS vendors. For simplicity, we focus on discussing the deployment incentives for the Internet registries and ASes, as past experiences of deploying DNSSEC [50] and IPv6 [36] suggest that they are often the deployment bottlenecks.

For the Internet registries, IPA achieves similar security benefits as previous work that requires a PKI [34, 38, 46, 56, 57], but has significantly lower deployment and management costs. This is because IPA uses the top-level DNSSEC infrastructure to bind an IP prefix to its owner's key. A DNSSEC-enabled registry need not deploy or manage any additional infrastructure to deploy IPA. Therefore, we believe that the Internet registries will have stronger incentives to deploy IPA than deploy a dedicated PKI required by previous work.

The IPA design also provides stronger deployment incentives for ASes than previous work, because ASes need not wait for the Internet registries to deploy a PKI and need not deploy additional certificate distribution infrastructures. Once the Internet registries have deployed IPA using DNSSEC, the top-level ASes that obtain IP prefixes directly from those registries can obtain immediate security benefits by distributing their IP prefix certificates in BGP and signing their BGP messages. These ASes will form a "club" to prevent prefix hijacking attacks within the club [26]. Using the IP prefix delegation hierarchy inferred in § 7.2.2, we find that such top-level ASes account for more than 78% of the total ASes. Once the top-level ASes have deployed IPA, their customers can obtain security benefits by adopting IPA, and so on. As the size of the protected club increases, the immediate security benefits that an adopter obtains also increase, which encourages more adopters, and can lead to a network effect of adoption [26].

7.4 Security Analysis

IPA bootstraps accountability with cryptography-based secure identifiers. Its security builds on the secrecy of private keys. The design stores private identity keys offline and uses periodic key rollovers to protect private keys. As long as the private keys remain secret, other security modules can use IPA to achieve accountable routing and forwarding, and DoS mitigation (§ 5).

The IPA design uses self-certifying AS identifiers. An AS may mint non-existent child AS identifiers by delegating sub-prefixes to those minted child ASes. However, because the minted identifiers are associated with sub-prefixes inside the AS's address space, the network can hold malicious ASes accountable by their address spaces to prevent them from evading traffic policing or gaining unfair shares of network resources (§ 5.3). An AS may inflate the AS path length in a BGP message by inserting the minted child AS identifiers, but it can achieve this goal by padding its own identifier in the message, which is a common BGP practice.

8 Related Work

The most related work in scope is the AIP architecture [20], which uses self-certifying identifiers as host addresses and domain identifiers. IPA retains the hierarchical IP addressing structure, but uses self-certifying AS identifiers. Unlike AIP, IPA's deployment does not require host re-numbering or trusted host hardware, but it requires the global root of trust of today's Internet (IANA) to continue to exist and function.

Public Key Infrastructures (PKIs) offer a hierarchical way to securely bind an identifier to a public key. Much existing work on secure routing, such as S-BGP [38], soBGP [57], psBGP [56], SPV [34], and Origin Authentication [46], requires the Internet registries to establish dedicated global PKIs to certify IP prefix ownerships or AS number ownerships. IPA obviates such requirements by using the existing top-level DNSSEC infrastructure to certify IP prefix allocations and using self-certifying identifiers as AS numbers. soBGP proposes to use a new type of BGP message to distribute various certificates in the routing system, while IPA uses a standard BGP extension to distribute IP prefix certificates.

The DNS CERT resource record (RR) [37] provides a generic way to store multiple types of certificates such as X.509, SPKI, and PGP with a DNS name. These certificates do not necessarily certify the DNS zone delegations, and hence do not certify IP prefix delegations. In contrast, IPA uses the Designated Signer and DNSKEY RRs rather than the CERT RR to map a reverse DNS zone delegation to an IP prefix delegation.

Simon et al. define network-layer accountability as traffic source identification and malicious traffic deterrence [54]. Their design assumes pairwise and transitive trust between ASes, and uses ingress filtering and an evil-bit in a packet header to stop DoS flooding traffic. However, if an AS within the trusted accountable group becomes compromised or malicious, it may fail to perform ingress filtering or set the evil-bit, rendering the design ineffective. IPA provides a similar form of accountability, but uses cryptography to establish accountability and is robust to malicious or compromised ASes.

An early version of IPA [58] outlines its main design modules. This work provides essential design details, an IPA prototype, and a comprehensive evaluation regarding

IPA's performance, adoptability, and security properties.

9 Conclusion

Lack of accountability makes the Internet vulnerable to many attacks, including source address spoofing, DoS flooding, prefix hijacking, and route forgery attacks. This work presents IPA, a design that bootstraps accountability in today's Internet with deployable and low-cost enhancements. IPA uses the top-level DNSSEC infrastructure to securely bind an IP prefix to an AS's public key and distributes these secure bindings using the routing system itself to lower deployment costs. We show that IPA enables a suite of security solutions [38, 43, 45] that collectively can combat the aforementioned network-layer attacks. We have presented the detailed IPA design, evaluated its performance, and shown that it is gradually deployable and provides stronger incentives for early adoption than previous proposals [34, 38, 46, 56, 57].

Acknowledgment

We thank Jeff Chase and the NSDI reviewers for their useful comments, and David Andersen for shepherding the paper. This work is supported in part by NSF awards CNS-0845858, CNS-1040043, and CNS-1017858.

References

[1] APNIC DNSSEC Service. http://www.apnic.net/services/services-apnic-provides/registration-services/dnssec.
[2] ARIN DNSSEC Deployment Plan. https://www.arin.net/resources/dnssec/index.html.
[3] BIND. https://www.isc.org/software/bind.
[4] DNSSEC Keys. http://www.ripe.net/dnssec-keys/index.html.
[5] DNSSEC Policy and Practice Statement. http://www.ripe.net/rs/reverse/dnssec/dps.html.
[6] DNSSEC Trust Anchors From ARIN. https://www.arin.net/resources/dnssec/trust_anchors.html.
[7] in-addr.arpa Transition. http://in-addr-transition.icann.org.
[8] Internet AS-level Topology on March 1st, 2011. http://irl.cs.ucla.edu/topology.
[9] IPv6 Support in BIND 9. http://www.bind9.net/manual/bind/9.3.2/Bv9ARM.ch04.html.
[10] M Root DNS Server. http://m.root-servers.org.
[11] Quagga Routing Suite. http://www.quagga.net.
[12] RADb: Routing Assets Database. http://www.radb.net.
[13] RIS Routing Beacons. http://www.ripe.net/projects/ris/docs/beacon.html.
[14] SecSpider the DNSSEC Monitoring Project. http://secspider.cs.ucla.edu.
[15] CIDR Report. http://www.cidr-report.org, 2006.
[16] DNS Survey: October 2009. http://dns.measurement-factory.com/surveys/200910.html, 2009.
[17] Cisco Patches Bug That Crashed 1 Percent of Internet. http://www.reuters.com/article/idUS418825996320100831, 2010.
[18] DNSSEC Signatures in Reverse DNS Zones Now Enabled. http://www.apnic.net/publications/news/2010/dnssec-signatures, 2010.
[19] Root DNSSEC Status Update, 2010-07-16. http://www.root-dnssec.org/2010/07/16/status-update-2010-07-16, 2010.
[20] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*, 2008.
[21] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033, 2005.
[22] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035, 2005.
[23] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034, 2005.
[24] J. Bennett and H. Zhang. Hierarchical Packet Fair Queueing Algorithms. *IEEE/ACM ToN*, 5(5), 1997.
[25] M. A. Brown. Pakistan Hijacks YouTube. http://www.renesys.com/blog/2008/02/pakistan-hijacks-youtube-1.shtml, 2008.
[26] H. Chan, D. Dash, A. Perrig, and H. Zhang. Modeling Adoptability of Secure BGP Protocols. In *ACM SIGCOMM*, 2006.
[27] M. Crawford. Binary Labels in the Domain Name System. RFC 2673, 1999.
[28] Deterlab. http://www.deterlab.net.
[29] DNS Deployment Initiative. http://www.dnssec-deployment.org.
[30] H. Eidnes, G. de Groot, and P. Vixie. Classless IN-ADDR.ARPA Delegation. RFC 2317, 1998.
[31] M. Feldman, C. Papadimitriou, J. Chuang, and I. Stoica. Free-riding and Whitewashing in Peer-to-Peer Systems. *IEEE JSAC*, 24(5):1010–1019, 2006.
[32] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. In *ACM Symposium on Operating Systems Principles*, 2007.
[33] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *USENIX/ACM NSDI*, 2005.
[34] Y. Hu, A. Perrig, and M. Sirbu. SPV: Secure Path Vector Routing for Securing BGP. In *ACM SIGCOMM*, 2004.
[35] Y.-C. Hu, D. McGrew, A. Perrig, B. Weis, and D. Wendlandt. (R)Evolutionary Bootstrapping of a Global PKI for Securing BGP. In *ACM HotNets-V*, 2006.
[36] G. Huston. Measuring IPv6 Deployment. http://www.internetac.org/wp-content/uploads/2010/02/apnic-v6-oecd1.pdf, 2009.
[37] S. Josefsson. Storing Certificates in the Domain Name System (DNS). RFC 4398, 2006.
[38] S. Kent, C. Lynn, and K. Seo. Secure Border Gateway Protocol (S-BGP). *IEEE JSAC*, 2000.
[39] O. Kolkman and R. Gieben. DNSSEC Operational Practices. RFC 4641, 2006.
[40] M. E. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the Internet. In *ACM SIGCOMM*, 2010.
[41] B. Lampson. Accountability and Freedom. http://research.microsoft.com/en-us/um/people/blampson/Slides/AccountabilityAndFreedomAbstract.htm, 2005.
[42] A. Li, X. Liu, and X. Yang. Dirty-Slate Accountable Internet Design. Technical Report 2010-07 (available at http://www.cs.duke.edu/nds/papers/ipa-tr.pdf), Duke University, 2010.
[43] X. Liu, A. Li, X. Yang, and D. Wetherall. Passport: Secure and Adoptable Source Authentication. In *USENIX/ACM NSDI*, 2008.
[44] X. Liu, X. Yang, and Y. Lu. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets. In *ACM SIGCOMM*, 2008.
[45] X. Liu, X. Yang, and Y. Xia. NetFence: Preventing Internet Denial of Service from Inside Out. In *ACM SIGCOMM*, 2010.
[46] P. McDaniel, W. Aiello, K. Butler, and J. Ioannidis. Origin Authentication in Interdomain Routing. *Computer Networks*, 50(16):2953–2980, 2006.
[47] P. Mockapetris. Domain Names – Concepts and Facilities. RFC 1034, 1987.
[48] J. Nazario. Estonian DDoS Attacks - A Summary to Date. http://asert.arbornetworks.com/2007/05/estonian-ddos-attacks-a-summary-to-date, 2007.
[49] J. Nazario. Georgia DDoS Attacks - A Quick Summary of Observations. http://asert.arbornetworks.com/2008/08/georgia-ddos-attacks-a-quick-summary-of-observations, 2008.
[50] E. Osterweil, M. Ryan, D. Massey, and L. Zhang. Quantifying the Operational Status of the DNSSEC Deployment. In *IMC*, 2008.
[51] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006.
[52] P. Roberts. Massive Denial Of Service Attack Severs Myanmar From Internet. http://threatpost.com/en_us/blogs/massive-denial-service-attack-severs-myanmar-internet-110310, 2010.
[53] RouteViews Project. http://www.routeviews.org.
[54] D. R. Simon, S. Agarwal, and D. A. Maltz. AS-based Accountability as a Cost-effective DDoS Defense. In *USENIX HotBots*, 2007.
[55] Q. Vohra and E. Chen. BGP Support for Four-octet AS Number Space. RFC 4893, 2007.
[56] T. Wan, E. Kranakis, and P. van Oorschot. Pretty Secure BGP (psBGP). In *NDSS*, 2005.
[57] R. White. Securing BGP Through Secure Origin BGP. *The Internet Protocol Journal*, 2003.
[58] X. Yang and X. Liu. Internet Protocol Made Accountable. In *ACM HotNets-VIII*, 2009.
[59] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *ACM SIGCOMM*, 2005.
[60] A. R. Yumerefendi and J. S. Chase. Strong Accountability for Network Storage. *ACM Transactions on Storage*, 3(3), 2007.

Privad: Practical Privacy in Online Advertising

Saikat Guha, Bin Cheng, Paul Francis
Microsoft Research India, and MPI-SWS
saikat@microsoft.com, {bcheng,francis}@mpi-sws.org

Abstract

Online advertising is a major economic force in the Internet today, funding a wide variety of websites and services. Today's deployments, however, erode privacy and degrade performance as browsers wait for ad networks to deliver ads. This paper presents Privad, an online advertising system designed to be faster and more private than existing systems while filling the practical market needs of targeted advertising: ads shown in web pages; targeting based on keywords, demographics, and interests; ranking based on auctions; view and click accounting; and defense against click-fraud. Privad occupies a point in the design space that strikes a balance between privacy and practical considerations. This paper presents the design of Privad, and analyzes the pros and cons of various design decisions. It provides an informal analysis of the privacy properties of Privad. Based on microbenchmarks and traces from a production advertising platform, it shows that Privad scales to present-day needs while simultaneously improving users' browsing experience and lowering infrastructure costs for the ad network. Finally, it reports on our implementation of Privad and deployment of over two thousand clients.

1 Introduction

Online advertising is a key economic driver in the Internet economy, funding a wide variety of websites and services. Internet advertisers increasingly work to provide more personalized advertising. Unfortunately, personalized online advertising comes at the price of individual privacy [23]. Privacy advocates would like to put an end to advertising models that violate privacy, and indeed have had some success with startups in the early stages of deployment [19]. On the other hand, they have had little success with the more entrenched ad brokers like Google and Yahoo! [11]. Arguably the reason why privacy advocates have failed here is that they offer no viable alternatives, and so the privacy solution they propose is effectively to end on-line advertising. This paper presents a practical and substantially more private online advertising system that attempts to offer that alternative.

To effect real change in the privacy of commercial advertising systems, we require that our design goals for Privad include commercial viability. This in turn requires that Privad:

1. is private enough that privacy advocacy groups[1] support it,
2. targets ads well enough to produce better click-through rates (or conversion rates, etc.) than current systems,
3. is as or less expensive to deploy than current systems, and
4. fits within the current business framework for online advertising, and therefore more likely has a viable business model. In particular, the interaction between Privad and end users, advertisers, and publishers, should not significantly change.

These goals are contradictory in nature, and much of the design challenge is finding the right balance of privacy and practicality. Although our arguments for scalability (goal 3) are strong and are buttressed by trace-based analysis, microbenchmarks, and deployment, we cannot definitively say that we have satisfied the other goals. While we hope to demonstrate better targeting through an experimental deployment (goal 2), this remains future work. The business model (goal 4) can ultimately only be demonstrated through a successful commercial deployment. While we have discussed our design with a number of privacy advocates, and have gotten favorable responses (goal 1), it is nevertheless hard to predict how they would react to a serious commercial deployment.

In practice we believe that a commercial deployment of Privad would be a constant balancing act between the goals listed above: the broker would gauge the reaction of privacy advocates, and strengthen or weaken privacy in response. In the absence of this commercial deployment and meaningful feedback from privacy advocates, our design assumes that privacy advocates will be hard to win over, and therefore favors privacy concerns over business concerns. In other words, our design attempts to produce the most private system possible within the constraint of achieving a merely *feasible* business model. In this paper, we nail down a design, present arguments as to why our practical goals are feasibly satisfied, and

[1] Private organizations like the Electronic Frontier Foundation (EFF) and the American Civil Liberties Union (ACLU), and government organizations like the Federal Trade Commission (FTC) and Europrise.

describe the security and scalability properties that our design ultimately achieves.

Privad preserves privacy by maintaining user profiles on the user's computer instead of in the cloud. A small amount of information necessarily leaves the user's computer: coarse-grained classes of ads a user is interested in, the ads the user has viewed or clicked on and the websites that carried the ads, and the ranking of ads for auctions. This information, however, is handled in such a way that no party can link it back to the individual user, or link together multiple pieces of information about the same user. An anonymizing proxy hides the user's network address, while encryption prevents the proxy from learning any user information. A trusted open-source reference monitor at the user's computer prevents any Personally Identifying Information (PII) other than network address from leaving the computer.

By contrast, current advertising systems, such as Google and Yahoo!, are in a deep architectural sense not private: they gather information about users and store it within their data centers. These systems do not lend themselves to being audited by privacy advocates or regulators. Users are essentially required to completely trust these systems to not do anything bad with the information. This trust can easily be violated, as for instance in a confirmed case where a Google employee spied on the accounts of four underage teens for months before the company was notified of the abuses [4].

Privad is considerably more private than current systems (though admittedly this is a low bar; we believe that privacy advocates will hold us to a much higher standard). Privad does not, for instance, require trust in any single organization. Additionally, Privad is designed to be auditable by third-parties. Most of this auditing is automatic, through the use of a simple reference monitor in the client. While Privad makes it much harder for an organization to gather private user information, Privad's privacy protocols are not bullet-proof (for instance with respect to collusion and covert channels), and so Privad allows the use of human-assisted or learning-based monitoring to detect misbehavior at the semantic level.

The anonymizing proxy (called *dealer*) is a significant change to the current business framework (goal 4). The dealer is run by an untrusted third-party organization, e.g. datacenter operators. We discuss in later sections the justification behind the dealer model, auditing mechanisms, and the feasibility of providing the service. We estimate the dealer's operating cost at around a cent per user per year (Section 4). This can easily be met with funding from privacy-advocates or levies on brokers.

The other significant change is client software on the users' computers. A key challenge, then, is incentivising deployment of this client software. Privad is not aimed for users that disable ads altogether. For users

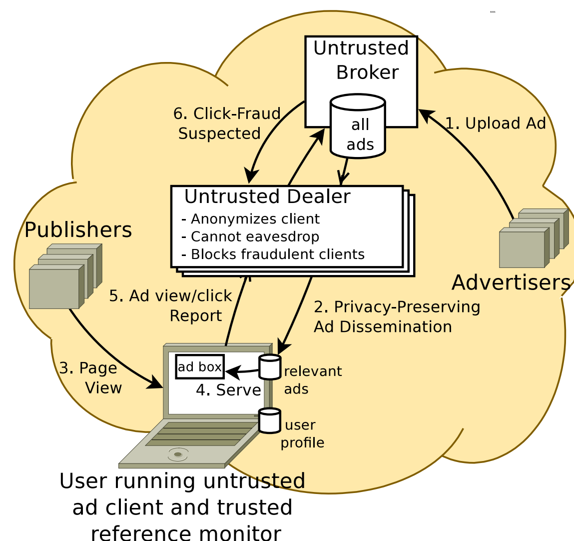


Figure 1: The Privad architecture

that do view and occasionally click ads, deploying requires first that Privad not degrade user experience in any way. We can ensure this by only showing ads in the same ad boxes that are common today (unlike previous adware, which employed disruptive advertising). Second, especially early on there must be some positive incentive for users to install it. This could be done through bundling other useful software, shopping discounts, or other incentives. Finally, it requires that privacy advocates endorse Privad. This at least prevents anti-virus software from actively removing the Privad client. Ideally, it even leads to privacy-conscious browser vendors (e.g. Firefox), anti-virus companies, or operating systems installing it by default.

The contributions of this paper are as follows: it presents a complete *practical* private advertising system. It describes the design of Privad, presents a feasibility study, and contributes a security analysis including both privacy and click-fraud aspects. It also gives a performance evaluation of our complete proof-of-concept implementation and pilot deployment of over two thousand users. Overall, Privad represents an argument that highly-targeted practical online advertising and good user-privacy are not mutually exclusive.

2 Privad Overview

There are six components in Privad: client software, client reference monitor, publisher, advertiser, broker, and dealer (see Figure 1). Publisher, advertiser, and broker all have analogs in today's advertising model, and play the same basic business roles. *Users* visit *publisher* webpages. *Advertisers* wish their ads to be shown to users on those webpages. The *broker* (e.g. Google) brings together advertisers, publishers, and users. For

each ad viewed or clicked, the advertiser pays the broker, and the broker pays the publisher.

There are three new key components for privacy in Privad. First, the task of profiling the user is done at the user's computer rather than at the broker. This is done by *client* software running on the user's computer. Second, all communication between the client and the broker is proxied anonymously by a kind of proxy called the *dealer*. The dealer also coordinates with the broker (using a protocol that protects user privacy) to identify and block clients participating in click-fraud. Finally, a thin trusted *reference monitor* between the client and the network ensures that the client conforms to the Privad protocol and provides a hook for auditing the client software. Encryption is used to prevent the dealer from seeing the contents of messages that pass between the client and the broker. The dealer prevents the broker from learning the client's identity or from linking separate messages from the same client.

At a high level, the operation of Privad goes as follows. The client software monitors user activity (for instance webpages seen by the user, personal information the user inputs into social networking sites, possibly even the contents of emails or chat sessions, and so on) and creates a user *profile* which contains a set of user *attributes*. These attributes consist of short-term and long-term *interests* and *demographics*. Interests include products or services like sports.tennis.racket or outdoor.lawn-care. Demographics include things like gender, age, salary, and location.

Advertisers submit ads to the broker, including the amount bid and the set of interests and demographics targeted by each ad. The client requests ads from the broker by anonymously subscribing to a broad interest category combined with a few broad non-sensitive demographics (gender, language, region). The broker transmits a set of ads matching that interest and demographics. These ads cover all other demographics and fine-grained locations within the region, and so are a superset of the ads that will ultimately be shown to the user. The client locally filters and caches these ads. If the user has multiple interests, there is a separate subscription for each interest, and privacy mechanisms prevent the broker from linking the separate subscriptions to the same user.

Ad auctions determine which ads are shown to the user and in what order. The ranking function, identical to the one used in industry today, uses in addition to the bid information, both user and global modifiers. User modifiers are based on things like how well the targeting information matches the user, and the user's past interest in similar ads. Global modifiers are based on the aggregate click-through-rate (CTR) observed for the ad, the quality of the advertiser webpage, etc.

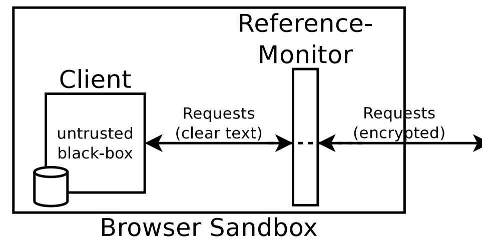


Figure 2: The Client framework

When the user browses a website that provides ad space, or runs an application like a game that includes ad space, the client selects an ad from the local cache and displays it in the ad space. A report of this *view* is anonymously transmitted to the broker via the dealer. If the user clicks on the ad, a report of this *click* is likewise anonymously transmitted to the broker. These reports identify the ad and the publisher on who's webpage or application the ad was shown. Privacy mechanisms prevent multiple reports from the same user from being linked together by the broker. The broker uses these reports to bill advertisers and pay publishers.

Unscrupulous users or compromised clients may launch click-fraud attacks on publishers, advertisers, or brokers. Both the broker and dealer are involved in detecting and mitigating these attacks (Section 3.4). When the broker detects an attack, it indicates to the dealer which reports relate to the attack. The dealer then traces these back to the clients responsible, and suppresses further reports from attacking clients, mitigating the attack.

Users, or privacy advocates operating on behalf of users, must be able to convince themselves that the client cannot *undetectably* leak private information. While having a trusted third-party write the client software might appear at first glance to be an option, it doesn't solve the problem — a trusted client simply moves the trust users place on brokers today to the third-party. At the same time, it requires brokers to make their trade-secret profiling algorithms known to the third party, and to parties auditing the client. Instead, Privad places a thin trusted reference monitor between the client and the network giving users and privacy advocates a hook to detect privacy violations (Section 3.5). It treats the client in a black-box manner (Figure 2), allowing the broker to use existing technological and legal frameworks for protecting trade-secret code. The reference monitor itself is simple, open source, and open to validation so its correctness can be verified, and can therefore be trusted by the user.

Note that Figure 1 does not portray the interaction that takes place between client and advertiser *after* an ad is clicked. For the purpose of this paper, we assume that a click brings the client directly to the advertiser as is the case today. We realize that this is a problem, because the finer-grained targeting of Privad gives unscrupulous ad-

vertisers more information than they get today. The Privad architecture leaves open the possibility of privately proxying the post-click session between client and advertiser, and even protecting the client from inadvertently releasing sensitive information. Because of space limitations, we do not further discuss this option, and only consider protecting the user from the broker and dealer. Privad does not modify today's relationship between client and publisher.

3 Privad Details

This section provides details on ad dissemination, ad auctions, view/click reporting, click-fraud defense and the reference monitor. It also puts forth some of the rationale for our design decisions. These details represent a snapshot of our current thinking. While ad dissemination, reporting, and reference monitor are quite stable, the click-fraud defense, and auctions may easily evolve as we do more analysis and testing. We present them here so as to present a complete argument for Privad's viability.

3.1 Ad Dissemination

The most privacy-preserving way to disseminate ads would be for the broker to transmit all ads to all clients. In this way, the broker would learn nothing about the clients. In [13], we measured Google search ads and concluded that there are too many ads and too much *ad churn* for this kind of broadcast to be practical. We observed that the number of impressions for ads is highly skewed: a small fraction of ads (10%) garner a disproportionate fraction of impressions (80%). Furthermore, this 10% of ads tend to be more broadly targeted and therefore of interest to many users. It may therefore be cost effective to disseminate only this small fraction of ads to all users, for instance using a BitTorrent-like mechanism. For the remaining 90%, however, a different approach is needed. We therefore design a privacy-preserving pub-sub mechanism between the broker and client to disseminate ads.

The pub-sub protocol (Figure 3) consists of a client's request to join a *channel* (defined below), followed by the broker serving a stream of ads to the client.

Each channel is defined by a single interest attribute and limited non-sensitive broad demographic attributes, for instance wide geographic region, gender, and language. The purpose of the additional demographics is to help scale the pub-sub system: limiting an interest by region or language greatly reduces the number of ads that need to be sent over a given channel while still maintaining a large number of users in that channel (in the k-anonymity sense). Channels are defined by the broker. The complete set of channels is known to all clients, for instance by having dealers host a copy (signed by

Figure 3: Message exchange for pub-sub ad dissemination. $E_x(M)$ represents the encryption of message M under key x. B is the public key of the broker. C is a symmetric key generated by the client for only this subscription.

the broker). A client joins a channel when its profile attributes match those of the channel.

The join request is encrypted with the broker's public key (B) and transmitted to the dealer. The request contains the pub-sub channel ($chan$), and a per-subscription symmetric key (C) generated by the client and used by the broker to encrypt the stream of ads sent to the client. The dealer generates for each subscription a unique (random) request ID (Rid). It stores a mapping between Rid and the client, and appends the Rid to the message forwarded to the broker. The broker attaches the Rid with ads published, which the dealer uses to lookup the intended client to forward the ads to.

The broker determines which ads should be sent and for how long they should be cached at the client. For instance, the broker stops sending ads for an advertiser when the advertiser nears his budget limit. Note that not all ads transmitted are appropriate for the user, and so may not be displayed to the user. For instance, an ad may be targeted towards a married person, while the user is single. Because the subscription does not specify marital status, the broker sends all ads independent of marital status or other targeting, and the client filters out those that do not match. Over time, the broker can estimate the number of ads that must be sent out for a particular advertiser to generate a target number of views and clicks.

3.2 Ad Auctions

Auctions determine which ads are shown to the user and in what order. For the advertiser, the auction provides a fair marketplace where the advertiser can influence the frequency and position of its ads through its bids. The broker additionally wants to maximize revenue, primarily by maximizing click-through rates (CTR). This is because most of today's advertising systems charge advertisers for clicks, not views. The broker also wants to minimize auction churn, generally by using a second-price auction [8]. A second-price auction is one whereby the bidder pays not the amount he bid, but the amount bid by the next lower bidder. This prevents the bidder from having to frequently change its bid in an attempt to probe for the bid value one unit higher than the next lower bidder.

Compared to today's brokers, which have full information about the system and can decide exactly which ads are shown where, in Privad both the client and the broker influence which ads are shown. This changes

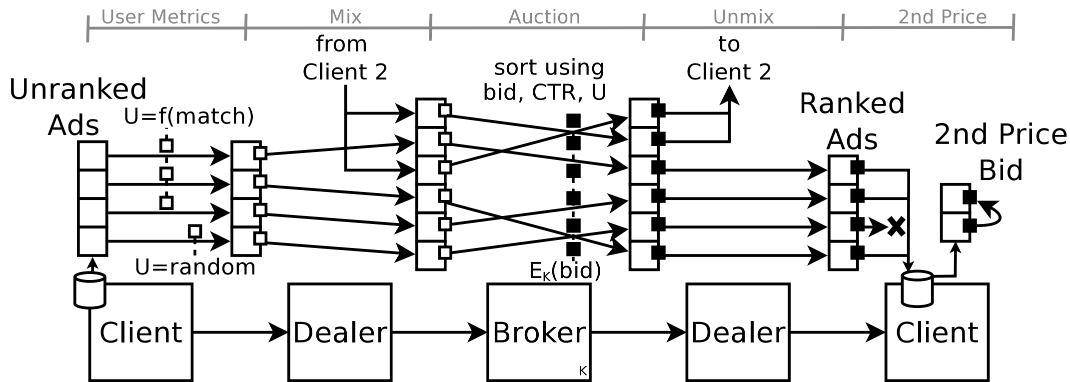


Figure 4: Industry-standard GSP Auction. Client annotates ads (across all channels) with quality of match, or random number if the ad doesn't match the user. Dealer mixes annotations from multiple clients. Broker ranks ads by bid, global click-through rate, advertiser quality, and match quality, and annotates the result with opaque bid information. Dealer slices auction result by client. Client filters out non-matching ads. Client reports encrypted second-price bid on click.

many aspects of the auction: for instance when the auction is run, over what set of ads, and the criteria by which second price is decided. The design space for Privad auctions is very large, and its complete exploration is a topic of further study. Nevertheless we describe two proof-of-concept auctions here.

A simple auction from this design space goes as follows. The broker periodically runs the auction over the set of ads targeted to a given pub-sub interest channel, producing a ranked set of ads. The ranking is preserved when ads are sent to clients. Clients filter out non-matching ads, slightly modify the ranking according to the quality of the demographic match for each ad, and show ads to users based on the modified ranking. When the broker receives a click report, it uses its original ranking to select the second price.

This auction is clearly different from Google's GSP auction [8]. For instance, with GSP, the auction is run when the browser requests a set of ads, and the second price is based on the ad below the clicked ad on the actual web page. We cannot necessarily say that our simple auction is worse than or better than GSP—this is a complex question and depends on, among other things, the evaluation criteria. As a demonstration of commercial viability, however, we now present a more complex auction that is identical to the industry-standard GSP auction mechanism.

In this second approach (Figure 4), the broker conducts the auction in a separate exchange. First, ads are sent to clients using pub-sub as originally described. The broker attaches a unique instance ID (Iid) to each copy of the ad published (not shown in figure). For each ad, the client computes a coarse score (U), typically between 1 and 5, as follows: for ads that match the user, the score reflects the quality of match with 5 signifying the best possible match. For ads that don't match the user, the score is a random number. To rank ads, the client sends (Iid, U) tuples for all ads in the client's database to the dealer. The dealer aggregates and mixes tuples for different clients before forwarding them to the broker. The broker ranks all the ads in the message. The ranking is based on both global and user modifiers (e.g. bids, CTR, advertiser quality, and client score). Note the ranked result contains all ads from the same client in the correct order, interspersed with ads for other clients (also in their correct order). The broker returns this ranked list to the dealer. The dealer uses the Iid to slice the list by client and forwards each slice to the appropriate client. The client discards the ads that do not match the user, and stores the rest in ranked order.

To obtain the GSP second price, the broker encrypts the bid information with a symmetric key (K) known only to the broker and sends it along with the ad. When a set of ads are chosen to be shown to the user, the client pairs up the encrypted bid information for ad $n + 1$ with that of ad n. This encrypted bid pair is sent as part of the click report, which the broker decrypts to determine what the advertiser should be charged.

3.3 View/Click Reporting

Ad views and clicks, as well as other ad-initiated user activity (purchase, registration, etc.) needs to be reported to the broker. The protocol for reporting ad events (Figure 5) is straightforward. The report containing the ad ID (Aid), publisher ID (Pid), and type of event (view, click, etc.) is encrypted with the broker's public-key and sent through the dealer to the broker. The dealer attaches a unique (random) request ID (Rid) and stores a mapping between the request ID and the client, which it uses later to trace suspected click-fraud reports in a privacy-preserving manner.

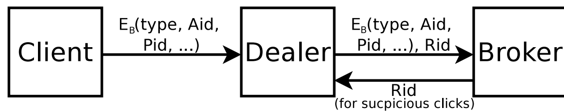


Figure 5: Message exchange for view/click reporting and blocking click-fraud. B is the public key of the broker. Aid identifies the ad. Pid identifies publisher website or application where the ad was shown. For second-price auctions, the opaque auction result is included. Rid uniquely identifies the report at the dealer.

3.4 Click-Fraud Defense

Click-fraud consists of users or bots clicking on ads for the purpose of attacking one or more parts of the system. It may be used to drive up a given advertiser's costs, or to drive up the revenue of a publisher. It can also be used to drive up the click-through-ratio of an advertiser so that that advertiser is more likely to win auctions.

Generally speaking, privacy makes click-fraud more challenging because clients are hidden from the broker. Privad addresses this challenge through an explicit privacy-preserving protocol between broker and dealer. Both the broker and dealer participate in detecting and blocking click-fraud; the dealer by measuring view and click volumes from clients, the broker by looking at overall click behavior for advertisers and publishers.

Blocking a fraudulent client once an attack is detected is straightforward. When a publisher or advertiser is under attack, the broker tells the dealer which report IDs are suspected as being involved in click-fraud. The dealer traces the report ID back to the client, and if the client is implicated more than some set threshold, subsequent reports from that client are blocked.

As with today's ad networks, there is no silver bullet for detecting click-fraud. And like ad networks today, the approach we take is *defense in depth* — a number of overlapping detection mechanisms (described below) operate in parallel; each detection mechanism can be fooled with some effort; but together, they raise the bar.

Per-User Thresholds. The dealer tracks the number of subscriptions, and the rates of view/click reports for each client (identified by their IP address). Clients that exceed thresholds set by the broker are flagged as suspicious. The broker may provide a list of NATed networks or public proxies so higher thresholds may apply to them.

Blacklist. Dealers flag clients on public blacklists, such as lists maintained by anti-virus vendors or network telescope operators that track IP addresses participating in a botnet. Dealers additionally share a blacklist of clients blocked at other dealers.

Honeyfarms. The broker operates honeyfarms that are vulnerable to botnet infection. Once infected, the broker can directly track which publishers or advertisers are under attack. When a report matching the attack signature is received, the broker asks the dealer to flag the originating client as suspicious.

Historical Statistics. The dealer and broker maintains respectively a number of per-client, and per-publisher and per-advertiser statistics including volume of view reports, and click-through rates. Any sudden increase in these statistics cause clients generating the reports to be flagged as suspicious.

Premium Clicks. Based on the insight behind [21], a user's purchase activity is used as an indication of honest behavior. Clicks from honest users command higher revenues. The broker informs the dealer which reports are purchases. The dealer flags the origin client as "premium" for some period of time, and attaches a single "premium bit" to subsequent reports from these clients.

Bait Ads. An approach we are actively investigating is something we term "bait ads" (similar to [14]), which can loosely be described as a cross between CAPTCHAs and the invisible-link approach to robot detection [27]. Basically, bait ads contain the targeting information of one ad, but the content (graphics, flash animation) of a completely different ad. For instance, a bait ad may advertise "dog collars" to "cat lovers". The broker expects a very small number of such ads to be clicked by humans. A bot clicking on ads, however, would unwittingly trigger the bait. It is hard for a bot to detect bait, which for image ads amounts to solving semantic CAPTCHAs (e.g. [9]). Bait ads are published by the broker just like normal ads. When a click for a bait ad is reported, the broker informs the dealer, which flags the client as potentially suspicious.

These mechanisms operate in concert as follows: per-user thresholds force the attacker to use a botnet. Honeyfarms help discover botnets, and blacklists limit the amount of time individual bots are of use to the attacker. Historical statistics block high-intensity attacks, instead forcing the attacker to gradually mount the attack, which buys additional time for honeyfarms and blacklists to kick in before significant financial damage is caused. At the same time, bait ads disseminated proactively can detect low volume attacks due to the strong signal generated by a relatively small number of clicks, while disseminated reactively, bait ads can reduce false positives. And finally, premium ads, by forcing the attacker to spend money to acquire and maintain "premium" status for each bot, apply significant economic pressure, which is magnified by bots being blacklisted.

Overall these mechanisms have the effect of more-or-less putting Privad back on an even footing with current ad networks as far as click-fraud is concerned.

3.5 Reference Monitor

The reference monitor has six functions geared towards making it difficult for the black-box client to leak pri-

vate information. We model the reference monitor on Google's Native Client (NaCl) sandbox [34] that allows running untrusted native code within a browser. As with NaCl, the sandbox presents a highly narrow and hardened API to untrusted code, and is itself open to validation by security experts and privacy advocates.

The reference monitor is hardened in at least the five following ways. First, the reference monitor validates that all messages in and out of the client follow Privad protocols. For this, the client is operated in a sandbox such that all network communication must go through the reference monitor in the clear (Figure 2). Second, it is the monitor that encrypts outbound messages from the client (and decrypts inbound messages). Third, the monitor is the source of all randomness in messages (e.g. session keys, randomized padding for encryption etc.). Fourth, the monitor may additionally provide cover traffic or introduce noise to protect user privacy in certain Privad operations. Fifth, the monitor arbitrarily delays messages or adds jitter to disrupt certain timing attacks.

Technological means for disrupting covert channels is, of course, not enough since the client may attempt to leak information through semantic means. For instance, the client might send lima-beans when it really means no-health-insurance. The sixth and final function of the reference monitor is therefore to provide an auditing hook, which can be used for instance to interpose a human-in-the-loop. Interested users may occasionally inspect messages for accuracy, and/or privacy advocates may set up honeyfarm clients, train them with specific profiles, and monitor them for inconsistent behavior using automated techniques presented in [12].

3.6 User Profiling

Even though the client is ultimately in charge of profiling the user, it can nevertheless leverage existing cloud-based crawlers and profilers through a privacy-preserving query mechanism. At a high level the query protocol is similar to the pub-sub protocol (Figure 3) operating as a single request-response pair; the request contains the website URL and the response contains profile attributes. Beyond this, the client can locally scrape and classify pages, incorporate social feedback, or even allow publisher websites to explicitly influence the profile. Overall, the user profiling options in Privad *adds to* existing cloud-based algorithms while preserving privacy, and therefore has the potential to target ads better than existing systems.

4 Feasibility

To validate the basic feasibility of Privad, we estimate worst-case network and storage overhead based on a trace of ads delivered by Microsoft's advertising platform (processing overhead is measured in Section 6).

Network and storage overhead at the client is due primarily to pub-sub ad dissemination. We use a trace of Bing search ads to determine an expected number of channels per client and ads per channel. We make the pessimistic assumption that all ads associated with a channel are transmitted to all subscriptions for that channel. We expect to be far more efficient than this in practice, since we can design our pub-sub service so that clients receive only fractionally more ads than necessary to fill their ad boxes (subject to k-anonymity and advertiser budget constraints). Summarizing our results, assuming compression and a 1MB local cache, we estimate the client will download less than 100kB per day on average (worst case: 20MB cache, 1.25MB daily download: less than a typical MP3 song). Even adjusting for the fact that our trace represents a good fraction, but a fraction nevertheless, of the search advertising market, and doesn't include contextual advertising, this load poses little concern.

We arrive at these estimates as follows: The Bing trace we used (for over 2M users in the USA sampled on Sep. 1, 2010) classifies users and ads into 128 interest categories. On average, each user is mapped to 2 interest categories on a given day (9 categories in the 99th percentile case). Using 2–4 coarse-grained geographic regions per state, we obtain several tens of thousand distinct interest-region-gender Privad channels. Remapping Bing ads to these channels results, on average, in slightly less than 2K ads for each channel (10K in the 99th percentile); note, an ad may be mapped to multiple channels. Each ad is roughly 250 bytes of text including the URL. This results in an average unoptimized daily download size of around 1MB (and less than 25MB in the worst case). Compressing ad content (in bulk) reduces download size by a factor of 10.

Of these, only the subset matching the user's other demographic attributes need to be stored in the client's local cache. Using the Bing trace's age-group classification alone, we get a factor of 5 reduction in storage. Occupation, education, marital-status etc. may further reduce storage requirements but we lack data to estimate these. Cached ad data can then be used to further reduce client network traffic. This requires a slight modification to the pub-sub protocol to periodically transfer a bitmap of active/inactive ads on the channel. Based on two weeks of trace data, we find that 54% of ads on a channel were seen the previous day (and around 70% within the previous 4 days; there is little added benefit for caching beyond 4 days). Thus with a warmed up 1MB cache, the client needs to download on average 100kB (1.25MB worst case) of compressed ad content plus a few tens of kilobytes of periodic bitmap data per day. Privad does not change the number of ads viewed by the user; based

on the Bing trace we estimate the client's upload traffic will be less than 20kB per day on average.

Consequently, we estimate the broker will send around 100kB and receive around 20kB per client per day, while the dealer acting as a proxy will send (and receive) around 120kB per client per day. While broker network overhead is more than today, the Privad broker trades-off network for lower processing overhead. There is, however, no simple comparison of Privad broker processing overhead with that of existing systems. Todays systems are *synchronous*: they request a small number of ads frequently, and ad selection plus auction plus ad delivery must occur in milliseconds. Privad is *asynchronous*: a large number of ads are requested infrequently, and these do not have to be delivered immediately (overhead quantified in Section 6). Thus comparing overall broker costs depends, among other factors, on the reduction in broker processing overhead and corresponding reduction in datacenter provisioning costs, versus bandwidth costs. As for the dealer, the network overhead works out to less than 88MB per user per year. Assuming the dealer leases datacenter resources at market prices, this amounts to less than $0.01 per user per year (based on current Amazon EC2 pricing [2]).

5 Implementation and Pilot Deployment

We have implemented the full Privad system and deployed it on a small scale. The system comprises a client implemented as a 210KB addon for the Firefox web browser, a dealer, and a broker. Out of the 11K total lines of code, the dealer consists of only 700 lines — well within limits of what can be manually audited.

We have deployed Privad with a small group of users comprised primarily of 2083 volunteers[2] we recruited using Amazon's Mechanical Turk service [1]. The primary purpose of the deployment is to convince ourselves that Privad represents a complete system. To this end the deployment exercises all aspects of Privad including user profiling (by scraping the user's Facebook profile and Google Ad Preferences), pub-sub ad dissemination, GSP auctions, view/click reporting, and basic click-fraud defense. For test ad data we scrape and re-publish Google ads through our system; since we lack targeting information for these ads, we target randomly. The system has been in continuous operation since Jan 1, 2010, with over 271K ads viewed and 238 ads clicked as of Jan 6, 2011.

The primary implementation challenge is the effort required to scrape webpages for profiling purposes. Facebook's and Google's layout changed on multiple occasions during our deployment, which required us to update the client code (using the addon's autoupdate mechanism). We are presently working on a higher-level language (and interpreter) for scraping webpages that will allow us to react more quickly to website changes.

[2]Users were offered an average one-time reward of $0.40 (for the 1 minute it took on average to install the addon) with mechanisms in place to prevent cheating. While users were required to leave the addon installed for at least a week to get paid, most users either forgot about it or chose to leave it installed for longer. As of Jan 6, 2011, 429 users still have the addon installed.

6 Experimental Evaluation

We use microbenchmarks to evaluate our system at scale.

Broker: We benchmark first the performance of subscribe and report messages at the broker since they involve public-key operations. Without optimizations, as expected, performance is bottlenecked by RSA decryptions. While crypto optimizations could be offloaded to hardware [18], since the broker is in any event untrusted, we additionally have the option of offloading to idle (untrusted) clients in the system (without impacting privacy guarantees). With this optimization, the broker needs only perform symmetric-key (AES) and hashing (SHA1) operations, which can be done at line speed using dedicated hardware [22]. Our software-based implementation achieved a throughput of 6K subscribe and report requests per second (on a single core of a 3GHz workstation), can publish 8.5K ads per second, and perform around 30K auctions per second. We note that request throughput in our broker is in the same ballpark as production systems today (based on the traces mentioned earlier); although this is somewhat of an apples-to-oranges comparison since brokers in Privad are much simpler.

In all cases the measured performance did not depend on the number of subscriptions or unique ads since all lookups at the broker are $\mathcal{O}(1)$; all runtime state (subscriptions, ads) is cached in memory and backed by persistent storage. The broker is designed with no shared state so it can trivially scale out to multiple cores.

Dealer: Our dealer can forward 15K requests per second (on the same hardware) in both directions, which is sufficient for handling nearly 200K online clients (based on request rates from our deployment). The bottleneck is due to client-side polling which arises from implementing Privad's asynchronous protocols on top of a request-response based transport (HTTP). With the emerging WebSockets standard [16], we believe we can eliminate this polling and support well over a million clients per dealer core.

Client: Finally we focus on how Privad improves a user's web browsing experience by eliminating network round-trips in the critical path of rendering webpages. Figure 6 compares Privad performance to existing ad networks. The figure compares the delay added for both populating ad boxes (on the 20 most popular sites as ranked by Alexa), and for completing the redirect to the advertiser webpage after a click. For Privad, we measured the time taken to populate ad boxes as we

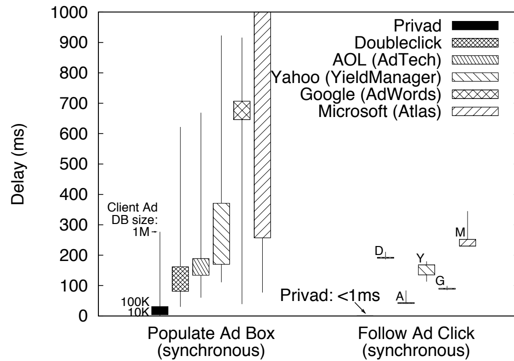


Figure 6: Privad eliminates network RTTs for showing ads, and reporting clicks. Whiskers for Privad show performance as the number of (relevant) ads in the client's database scales to 1 million. Whiskers and boxes for existing ad networks show minimum and maximum latencies, and quartiles.

scale the number of (relevant) ads cached in the client database. As mentioned, we estimate the typical number of cached ads to be between 10K (average) to 100K (worst case); we benchmark with a factor of ten margin. As one might expect, our client implementation outperforms existing ad networks since displaying ads requires only local disk access. Our client can populate ad boxes, based on keywords or website context, in 31ms. In existing networks, we found the delay was dominated by the ad selection process; downloading the actual ad content (e.g. 30kB flash file) took less than 2ms. Doubleclick, which to our knowledge does not perform demographic or context sensitive advertising, took 129ms in the median case, and Google, which does perform context sensitive advertising, took 670ms. With regards to reporting clicks, existing ad networks must perform a synchronous redirect through the ad network, which consumes several RTTs. Since Privad reports clicks asynchronously (when browser is idle), the redirect is unnecessary, thus allowing much faster advertiser page-loads.

Our client scrapes webpages, pre-fetches ads, conducts auctions, and sends reports in the background. Messages that require public-key encryptions take between 68ms (on a workstation) to 160ms (on a netbook) to construct, but since they are performed when the browser is idle, they are imperceptible to the user. The client uses negligible memory since ads are stored on disk; there is no appreciable change in the browser's memory footprint whether the client is enabled or disabled. During our 12 month deployment, we have not received any negative feedback, performance related or otherwise, from users[3].

[3] or, for that matter, positive feedback.

7 Privacy Analysis

Broadly speaking, Privad uses technological means to protect user privacy. Privad provides privacy through *unlinkability* [28] (described below), and uses the dealer mechanism to ensure this. It is worth considering briefly alternative design points that we opted against.

Considering it is believed to be impossible to design systems that are secure against covert channels and collusion [17, 26], neither we, nor privacy advocates expect bulletproof privacy. Privacy advocates instead have the much softer requirement that "individuals [be] able to control their personal information", and if privacy is violated, the ability to "hold accountable organizations [responsible]" [5]. Privad trivially satisfies the first requirement by storing all personal information on the user's computer and assuring unlinkability. In the absence of covert channels or collusion, this prevents any organization from learning about users, thereby preventing privacy violations in the first place. In the presence of covert channels or collusion, the organization's willing and explicit circumvention of technological privacy safeguards strongly implies malicious intent (in the legal sense) to which they can be held accountable.

As a result, the oversight task for privacy advocates is reduced from detecting any kind of privacy violation, including those purely internal to a broker, to detecting collusion and the use of covert channels. As we discuss below, Privad incorporates existing (and future) techniques to disrupt or detect covert channels through the reference monitor mechanism and careful protocol design. Detecting collusion is easier with the dealer mechanism as compared to, say, a mixnet like TOR [6]. Not only does TOR not meet business needs by giving up any visibility into click fraud, TOR's threat model is a poor match for Privad since a single entry node colluding with the broker can compromise the anonymity of all users connecting through that node [3]. In contrast to mixnet nodes, a dealer organization (e.g. datacenter operators) can be contractually bound, and its non-collusionary involvement be monitored by privacy advocates. This model is in use today and is approved for instance by the European privacy certification organization Europrise [10].

Given that Privad relies to an extent on accountability, one might ask why a purely regulatory solution doesn't suffice. There are two problems. First, entrenched players like Google have strong incentives, lobbying power, and the capital needed to maintain the status quo. Indeed many parallels can be drawn to the network-neutrality battle where powerful ISPs successfully resisted new regulations threatening their business model [33]. Second, even if regulations were passed, enforcement would require third-party auditing of all broker operations, which is impractical due to the complexity and scale of these systems. Market forces, such as

competition from a startup offering better ROI to advertisers through deeper personalization (with backing from privacy advocates), can arguably effect change more easily.

In the remainder of this section we first define informally what we mean by user privacy and our trust assumptions. We then address the technical measures pertaining to covert channels. We then consider a series of attacks on the system, the defense to the attack, and a discussion of the extent to which the defense truly solves the attack.

7.1 Defining Privacy

Our privacy goals are based on Pfitzmann and Köhntopp's definition of anonymity [28] which is unlinkability of an *item of interest* (IOI) and some logical user identifier. Privad has three types of IOI; IP address, and interest attributes and demographic attributes. Pfitzmann and Köhntopp consider anonymity in terms of an *anonymity set*, which is the set of users that share the given item of interest — the larger this set, the "better" the anonymity. Personally Identifiable Information (PII) is information for which the anonymity set comprises a single (or a very small number of) elements; e.g., the IP address is PII. Examples of non-PII anonymity sets in Privad include: the set of users that join a pub-sub channel, the set of users that visit a given publisher, and the set of users that view or click a given ad (i.e. probably share some or all of the ad's attributes).

In our definition of privacy we draw a distinction between IOI that contain PII and IOI that do not, as follows:

P1) Profile Anonymity: No single player can link any PII for a user with any attribute in the user's profile.

P2) Profile Unlinkability: No single player can link together more than a threshold number of (non-PII) profile attributes for the same user, which would otherwise allow them to, over time, construct a unique profile that could be deanonymized using external databases.

Existing ad networks, of course, satisfy neither Profile Anonymity nor Profile Unlinkability.

Note that for Profile Unlinkability we use "number of profile attributes" rather than the size of the anonymity set even though the former doesn't per se map directly onto the latter. Different attributes imply different sizes of anonymity sets (e.g., music vs. sports.skiing.cross-country). Ideally, Privad would dynamically guarantee a minimum anonymity set size at runtime, but this is not possible because any such approach is easily attacked with Sybils [7], e.g. a botnet of clients masquerading as members of that set. It is possible, however, to estimate offline the rough expected anonymity set size for an attribute with outside semantic knowledge.

The approach towards privacy in Privad is then as follows: 1) offline semantic analysis by privacy advocates establishes per-message thresholds for Profile Unlinkability; this is enforced at runtime by the monitor as we discuss later in Attack A9. 2) Mechanisms in Privad ensure multiple messages from the same client cannot be linked together, and therefore the system as a whole cannot violate Profile Unlinkability. And 3) since the dealer is the only party that learns PII (IP address) and nothing else about the user, Profile Anonymity is trivially satisfied.

7.2 Trust Assumptions

The user trusts only the reference monitor; the client software, dealer and broker are all untrusted. Privacy advocates are expected to play a watchdog role by validating the reference monitor, monitoring dealer operation, and running honeyfarms to detect covert channels. The broker does not trust clients, dealers, or reference monitors. Attack A4 below discusses malicious dealers including those that may engage in click fraud. Privad does not modify any interactions users or brokers have with publishers or advertisers. The advertiser and publisher, like today, can see the user's browsing behavior on their own site, and trust the broker to perform accurate billing.

7.3 Covert Channels

A malicious broker may distribute a malicious client that attempts to leak data using covert channels. The bandwidth of covert channels is reduced by bounding non-determinism in messages. Note first of all that the covert channel must come from Privad application message fields, not encapsulating protocol fields such as those in the crypto messages. This is because it is the reference monitor that takes care of crypto and message delivery functions. In addition, it is also the monitor that generates the one-time shared keys (for subscriptions) which otherwise represent the best covert channel opportunity.

Note next that the values of most message fields are driven by user behavior (outside client-control) and are subject to audit by privacy advocates or users. This includes the channel ID in subscriptions, and the type, publisher ID, and ad ID in reports, which together compose all remaining bits in subscribe and report messages. The next best opportunity for a covert channel would come from the user score in the GSP auction message (Figure 4). That is because this is the only client-controlled message field, albeit only 2 or 3 bits in size since the user score need only be in a small range. This bounds the information that can be leaked by a single message.

The Privad protocol and reference monitor make it hard to construct a covert channel across multiple messages. Since messages from the same source cannot, by design, be linked based on content, the attacker must use

some time-based watermarking technique (e.g., [32]). The reference monitor adds arbitrary delay or jitter to messages to disrupt such attempts. For this reason, all Privad protocols are designed to be asynchronous and use soft-state without any acknowledgments.

A computer system cannot completely close all covert channels, but by at least making it possible for privacy-advocates to detect them, and by establishing malicious intent by requiring attackers to circumvent multiple technical hurdles, Privad significantly increases the risk of being caught and thus decreases the utility of covert channels. This is in contrast to today where third-parties can neither detect privacy-violations, nor establish intent when violations are revealed [29].

7.4 Attacks and Defenses

This section outlines a set of key attacks on user privacy. Space constraints prevent us from discussing in detail attacks on advertiser and broker privacy. We do however briefly note the following. Broker privacy, in the form of trade secrets for profiling mechanisms, is maintained because client software is a black-box that does not need to be audited; and the broker can use the same legal and technical mechanisms used by desktop software companies today. Advertiser privacy is weakened because it is slightly easier to learn an ad's targeting information as compared to today's systems. Privad does not however change the ease with which an attacker can learn an advertiser's bids.

7.4.1 Attacker at Client

Attack A1: The attacker installs malware on a user's computer which provides the profile information to the attacker or otherwise exploits it.

Defense D1: Privad does not protect against malware reading the profile it generates. Our general stance is that even without Privad, malware today can learn anything the client is able to learn, and so not protecting against this threat does not qualitatively change anything. Having said that, obviously the existence of the profile does make the job of malware easier. It saves the malware from having to write its own profiling mechanisms. It also allows the malware to learn the profile more quickly since it doesn't have to monitor the user over time to build up the profile.

Ultimately what goes into the profile is a policy question that privacy advocates and society need to answer. Clearly information like credit card number, passwords, and the like have no place in the profile (though malware can of course get at this information anyway). Whether a user has AIDS probably also does not belong there. Whether a user is interested in AIDS medication, however, arguably may belong in the profile.

Indeed, there are pros and cons to keeping profile contents open. On the pro side, this makes it easier for privacy advocates to monitor the client and to an extent broker operation. On the con side, it makes life easier for malware. One option, if the operating system supports it, is to make the profile available only to the client process (e.g. through for instance SELinux [25]). This would protect against userspace malware, but not rootkits that compromise the OS. Another option is to leverage trusted hardware (e.g. [31]) when available. How best to handle the profile from this perspective is both an ongoing research question and a policy question.

7.4.2 Attacker at Dealer

A2: The attacker attempts to learn user profile information by reading messages at the dealer.

D2: The dealer proxies five kinds of messages: subscribe, publish, auction request and response, and reports. Of these, the dealer cannot inspect the contents of subscribe, report, and publish messages since the first two are encrypted with the broker's public key, and the last is encrypted with a symmetric key that is exchanged via the encrypted subscribe message. Auction messages, which are unencrypted, contain a random single-use Iid that identifies the ad at the broker and the client (exchanged over the encrypted publish message), but is meaningless to the dealer.

A3: The attacker injects messages at the dealer in order to learn a user's profile information.

D3: The dealer cannot inject a fake publish message since it would not validate at the client after decryption. If the dealer injects a fake subscribe message, all resulting publish messages would be discarded by the client since the client would not have a record of the subscribe or the associated key. The dealer cannot inject fake auction messages since the client would not have a record of the Iid. The dealer could reorder the auction result, but would not learn which ad the client viewed or clicked since reports are encrypted. The dealer injecting fake reports has no impact on the client; it is, however, identical to dealer-assisted click-fraud, which we consider next.

A4: The dealer itself engages in click-fraud, or otherwise does not comply with the broker's request to block fraudulent clients.

D4: The broker can independently audit that the dealer is operating as expected both actively and passively. The broker can passively track view/click volumes, and historical statistics on a per-dealer basis to identify anomalous dealers. Additionally the broker can passively monitor the rate of fraudulent clicks (e.g. using bait ads) on a per-dealer basis. The broker can detect suspicious dealer behavior if after directing dealers to stem a particular attack the rate of fraudulent clicks through one dealer does not drop (or drops proportionally less) than

for other dealers. Finally, the broker can actively test a dealer by launching a fake click-fraud attack from fake clients, and ensuring the dealer blocks them as directed.

A5: A particularly sneaky attack aimed at learning which users send view or click reports for a given publisher (or advertiser) is as follows. The dealer first launches a click-fraud attack on the given publisher (or advertiser). The broker identifies the attack. When a user sends a legitimate report for that publisher (or advertiser), the broker mistakenly suspects the report as fraudulent and asks the dealer to block the client. The dealer can now infer that the encrypted report it proxied must have matched the attack signature it helped create.

D5: First note that this attack applies only in the scenario where there are no other click-fraud attacks taking place other than the one controlled by the dealer (and the dealer somehow knows this). As part of the Privad protocol (Figure 5), however, the dealer does not learn how many attacks are taking place (even if there is only one ongoing attack), or which publishers or advertisers are under attack, or which attack the client was implicated in. Thus there is too much noise for the dealer to reach any conclusions about implicated clients.

7.4.3 Attacker at Broker

A6: The broker attempts to link multiple messages from the same user using passive or active approaches.

D6: We are only concerned with subscribe and reports messages since the dealer mixes auction requests. Privad messages do not contain any PII, unique identifier, or sequence number. The monitor ensures the per-subscription symmetric keys are unique and random. Additionally, the monitor disrupts timing based correlation, for instance by staggering bursts of messages (e.g. when the client starts up, or views a website with many adboxes). Altogether these defenses prevent the broker from linking two subscriptions, or two reports from the same user.

The broker may attempt to link a report with a subscription. The only way to do this is by publishing an ad with a unique ad ID, and waiting for a report with that ID. Privacy advocates can detect this by running honeyfarms of identical clients and ensuring ad IDs are repeated.

A7: During the GSP auction mechanism the broker attempts to link two ads published to the same client through different pub-sub subscriptions, thereby effectively linking two subscriptions.

D7: The property of the mix constructed at the dealer is such that tuples from the same client but for ads on different pub-sub channels are indistinguishable from tuples from two different clients each subscribed to one of the channels. The pub-sub protocol provides the same property. Thus the broker doesn't learn anything new from the auction protocol.

Note the broker can obviously link which ads it sent for the same subscription, but cannot determine which of them actually matched the user. This is because the client submits all ads received on a channel for auction whether or not it matched the user (enforced by the monitor); bogus user scores for non-matching ads prevents the broker from distinguishing between the two.

A8: The broker masquerades as a dealer and hijacks the client's messages thus learning the client's IP address. Possible methods of hijacking the traffic may include subverting DNS or BGP.

D8: The solution is to require Transport Layer Security (TLS) between client and dealer, and to use a trusted certificate authority. The reference monitor can insure that this is done correctly.

A9: The broker creates a channel with a large enough number of attributes that an individual user is uniquely defined. When that user joins the channel, the broker knows that a user with those attributes exists. This could be done for instance to discover the whereabouts of a known person or to discover additional attributes of a known person. For instance, if n attributes are known to uniquely define the person, then any additional attributes associated with a joined channel can be discovered.

D9: It is precisely for this reason that pub-sub channels definitions are static, well-known, and public (Section 3.1). Privacy advocates can look at channel definitions and ensure they meet a minimum expected anonymity set size. Additionally, the monitor can filter out channel definitions when the attributes for that channel exceed some set threshold.

Similar restrictions apply to the set of profile attributes an ad can target, with one difference. In the context of second-price auctions, the broker needs to necessarily link adjacent ads. Thus the monitor needs to enforce that the sum of attributes of the two ads involved in a click-report is below the threshold.

Note the ability to link two ads applies only to clicks. View reports do not contain second price information since otherwise a page with many ads would allow the broker to link each consecutive pair of ads, and therefore a whole chain of ads. While the same problem exists if the user were to click on the whole chain of ads, since clicks are rare this is not a big concern.

8 Related Work

There is surprising little past work on the design of private advertising systems, and what work there is tends to focus on isolated problems rather than a complete system like Privad. This related work section focuses only on systems that target private advertising per se, and mainly concentrates on the privacy aspects of those systems. In particular, we look at Juels [20], Adnostic [30], and Nurikabe [24].

Juels by far predates the other work cited here, and indeed is contemporary with the first examples of the modern advertising model (i.e. keyword-based bidding). As such, Juels focuses on the private distribution of ads and does not consider other aspects such as view-and-click reporting or auctions. Privad's dissemination model is similar to Juels' in that a client requests relevant ads which are then delivered. Indeed, Juels' trust model is stronger than Privad's. Juels proposes a full mixnet between client and broker, thus effectively overcoming collusion. We believe this trust model is overkill, and that his system pays for this both in terms of efficiency and in the mixnet's inability to aid the broker in click fraud.

Like Juels and Privad, Adnostic also proposes client-side software that profiles and protects user privacy. When a user visits a webpage containing an adbox, the URL of the webpage is sent to the broker as is done today. The broker selects a group of ads that fit well with the ad page (they recommend 30), and sends all of them to the client. The client then selects the most appropriate ad to show the user. The novel aspect of Adnostic is how to report which ad was viewed without revealing this to the broker. Adnostic uses homomorphic encryption and efficient zero-knowledge proofs to allow the broker to reliably add up the number of views for each ad without knowing the results (which remain encrypted). Instead, they send the results to a trusted third-party which decrypts them and returns the totals. By contrast to views, Adnostic treats clicks the same as current ad networks: the client reports clicks directly to the broker.

The privacy model proposed by Adnostic is much weaker than that of Privad. Privad considers users' web browsing behavior and click behavior to be private, Adnostic does not. Indeed, we would argue that the knowledge that Adnostic provides to the broker allows it to very effectively profile the user. A user's web browsing behavior says a lot about the user interests and many demographics. Knowledge of which ads a user has clicked on, and the demographics to which that ad was targeted, allow the broker to even more effectively profile the user. Finally, the user's IP address provides location demographics and effectively allows the broker to identify the user. Adnostic's trust model for the broker is basically honest-and-*not*-curious. If that is the case, then today's advertising model should be just fine.

Nurikabe also proposes client-side software that profiles the user and keeps the profile secret. With Nurikabe, the full set of ads are downloaded into the client. The client shows ads as appropriate. Before clicking any ads, the client requests a small number of click tokens from the broker. These tokens contain a blind signature, thus allowing the tokens to later be validated at the broker without the broker knowing who it previously gave the token to. The user clicks on an ad, the click report is sent to the advertiser along with the token. The advertiser sends the token to the broker, who validates it, and this validation is returned to the client via the advertiser.

Nurikabe has an interesting privacy model. They argue that, since the advertiser anyway is going to see the click, there is no loss of privacy by having the advertiser proxy the click token. By taking this position, Nurikabe avoids the need for a separate dealer. Our problem with this approach is that Nurikabe basically gives up on the problem of privacy from the advertiser altogether. It cannot report views without exposing this to the advertiser, thus reducing user privacy from the advertiser even more than today. View reporting is important, in part because it allows the advertiser to compute the CTR and know how well its ad campaign is going. Nurikabe also gives up any visibility into click fraud. Nurikabe mitigates click fraud only by rate limiting the tokens it gives to every user. As a result, the attacker need only Sybil itself behind a botnet and solve CAPTCHAs to launch a massive click-fraud attack which cannot be defended. Finally, in [13] the authors find through ad measurements that there are simply far too many ads (with too much churn) to be able to distribute them all to all clients.

Some aspects of Privad have previously been explored in [13, 15]. The seed idea behind Privad was planted in [15], a short paper revisiting the economic case for advertising agents on the endhost (i.e., distinguishing "adware" from "badware"), which presents a rough sketch of privacy-aware click reporting. In [13] we use measurement data to guide our design and explore the feasibility of building such a system. This paper presents the resulting detailed design, experimental evaluation, and security analysis of a full advertising system.

9 Summary and Future Directions

This paper describes a practical private advertising system, Privad, which attempts to provide substantially better privacy while still fitting into today's advertising business model. We have designs and detailed privacy analysis for all major components: ad delivery and reporting, click fraud defense, advertiser auctions, user profiling, and optimizations for scalability.

We are actively working on getting a better understanding of a number of Privad components. Foremost among these are how best to do profiling, how best to run auctions, the bait approach to click-fraud, and privacy from the advertiser. Another important problem is how to allow brokers and advertisers to gather rich statistical information about user behavior in a privacy-preserving way. Towards this end, we are looking at distributed forms of differential privacy. We are also working with application developers to deploy at Internet scale to give researchers a platform for experimenting with real users and advertisements.

Besides pursuing the technical aspects of Privad, we have discussed Privad with a number of privacy advocates and policy makers, and have applied for a Europrise privacy seal. We hope that Privad and other recently proposed private advertising systems spur a rich debate among researchers and privacy advocates as to the best ways to do private advertising, the pros and cons of the various systems, and how best to move private advertising forward in society.

References

[1] Amazon Mechanical Turk. `http://www.mturk.com`.

[2] Amazon Inc. Amazon Elastic Compute Cloud (Amazon EC2), Sept. 2010. `http://aws.amazon.com/ec2/`.

[3] K. Bauer, D. McCoy, D. Grunwald, T. Kohno, and D. Sicker. Low-Resource Routing Attacks Against Tor. In *Proceedings of the 2007 Workshop on Privacy in the Electronic Society (WPES)*, Alexandria, VA, Oct. 2007.

[4] A. Chen. GCreep: Google Engineer Stalked Teens, Spied on Chats. Sept. 2010. `http://gawker.com/5637234`.

[5] J. Chester, S. Grant, J. Kelsey, J. Simpson, L. Tien, M. Ngo, B. Givens, E. Hendricks, A. Fazlullah, and P. Dixon. Letter to the House Committee on Energy and Commerce. `http://tinyurl.com/y85h98g`, Sept. 2009.

[6] R. Dingledine, N. Mathewson, and P. Syverson. TOR: The Second-Generation Onion Router. In *Proceesings of USENIX Security Symposium '04*.

[7] J. R. Douceur. The Sybil Attack. In *Proceedings of IPTPS '02*.

[8] B. Edelman, M. Benjamin, and M. Schwarz. Internet Advertising and the Generalized Second-Price Auction: Selling Billions of Dollars Worth of Keywords. *American Economic Review*, 97(1):242–259, Mar. 2007.

[9] J. Elson, J. R. Douceur, J. Howell, and J. Saul. Asirra: A CAPTCHA that Exploits Interest-Aligned Manual Image Categorization. In *Proceedings of CCS '07*.

[10] Europrise. European Privacy Seal DE-080006p. `http://tinyurl.com/2dckmpx`.

[11] G. Gross. FTC Sticks With Online Advertising Self-regulation. *IDG News Service*, Feb. 2009.

[12] S. Guha, B. Cheng, and P. Francis. Challenges in Measuring Online Advertising Systems. In *Proceedings of IMC '10*.

[13] S. Guha, A. Reznichenko, K. Tang, H. Haddadi, and P. Francis. Serving Ads from localhost for Performance, Privacy, and Profit. In *Proceedings of HotNets '09*.

[14] H. Haddadi. Fighting Online Click-Fraud Using Bluff Ads. *SIGCOMM CCR*, 40(2):22–25, Apr. 2010.

[15] H. Haddadi, S. Guha, and P. Francis. Not All Adware is Badware : Towards Privacy-Aware Advertising. In *Proceedings of 9th IFIP conference on e-Business, e-Services, and e-Society*, Nancy, France, Sept. 2009.

[16] I. Hickson. The WebSocket API. `http://dev.w3.org/html5/websockets/`.

[17] N. Hopper, J. Langford, and L. V. Ahn. Provably Secure Steganography. In *Proceedings of Crypto '02*.

[18] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of NSDI '11*.

[19] A. Jesdanun. Ad Targeting Based on ISP Tracking Now in Doubt. *Associated Press*, Sept. 2008.

[20] A. Juels. Targeted Advertising ... And Privacy Too. In *Proceedings of the 2001 Conference on Topics in Cryptology*, pages 408–424, London, UK, 2001.

[21] A. Juels, S. Stamm, and M. Jakobsson. Combating Click Fraud via Premium Clicks. In *Proceesings of USENIX Security Symposium '07*, pages 1–10.

[22] M. Kounavis, X. Kang, K. Grewal, M. Eszenyi, S. Gueron, and D. Durham. Encrypting the Internet. In *Proceesings of SIGCOMM '10*.

[23] B. Krishnamurthy and C. E. Wills. Cat and Mouse: Content Delivery Tradeoffs in Web Access. In *Proceedings of WWW '06*.

[24] D. Levin, B. Bhattacharjee, J. R. Douceur, J. R. Lorch, J. Mickens, and T. Moscibroda. Nurikabe: Private yet Accountable Targeted Advertising. Under submission. Contact `johndo@microsoft.com` for copy, 2009.

[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*, Boston, MA, June 2001.

[26] I. S. Moskowitz and M. H. Kang. Covert Channels - Here to Stay? In *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS)*, pages 235–243, Gaithersburg, MD, July 1994.

[27] K. Park, V. S. Pai, K.-W. Lee, and S. Calo. Securing Web Service by Automatic Robot Detection. In *Proceesings of USENIX Annual Technical Conference '06*.

[28] A. Pfitzmann and M. Köhntopp. Anonymity, Unobservability, and Pseudonymity — A Proposal for Terminology. *Designing Privacy Enhancing Technologies*, 2001.

[29] B. Stone. Google Says It Inadvertently Collected Personal Data. *The New York Times*, May 2010. `http://tinyurl.com/2946cql`.

[30] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of NDSS '10*.

[31] Trusted Computing Group. TPM Specification Version 1.2. `http://www.trustedcomputinggroup.org/`.

[32] X. Wang, S. Chen, and S. Jajodia. Tracking Anonymous Peer-to-Peer VoIP Calls on the Internet. In *Proceedings of CCS '05*.

[33] E. Wyatt. U.S. Court Curbs F.C.C. Authority on Web Traffic. *The New York Times*, Apr. 2010. `http://tinyurl.com/yamowhd`.

[34] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, , and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of Oakland '09*.

Bazaar: Strengthening user reputations in online marketplaces

Ansley Post[†‡] Vijit Shah[*] Alan Mislove[*]

[*]*Northeastern University* [†]*MPI-SWS* [‡]*Rice University*

Abstract

Online marketplaces are now a popular way for users to buy and sell goods over the Internet. On these sites, user reputations—based on feedback from other users concerning prior transactions—are used to assess the likely trustworthiness of users. However, because accounts are often free to obtain, user reputations are subject to manipulation through white-washing, Sybil attacks, and user collusion. This manipulation leads to wasted time and significant monetary losses for defrauded users, and ultimately undermines the usefulness of the online marketplace.

In this paper, we propose Bazaar, a system that addresses the limitations of existing online marketplace reputation systems. Bazaar calculates user reputations using a max-flow-based technique over the network formed from prior successful transactions, thereby limiting reputation manipulation. Unlike existing approaches, Bazaar provides strict bounds on the amount of fraud that malicious users can conduct, regardless of the number of identities they create. An evaluation based on a trace taken from a real-world online marketplace demonstrates that Bazaar is able to bound the amount of fraud in practice, while only rarely impacting non-malicious users.

1 Introduction

Online marketplaces like eBay, Overstock Auctions, and Amazon Marketplace enable buyers and sellers to connect regardless of each other's location, allowing even the most esoteric of products to find a market. These marketplaces have greatly expanded the set of people who can act as a buyer or seller and, thus, can be viewed as democratizing commerce. These sites are extremely popular with users; in 2009, over $60 billion worth of goods was exchanged on eBay alone.

This new freedom, however, does not come without challenges. Online marketplaces are known to suffer from fraud, and often rely on user reputations—formed from the feedback provided by other users—in an effort to mitigate the effects of malicious activities on their sites. For example, on eBay, potential buyers often examine the reputation of the seller to determine the seller's trustworthiness. In fact, it has been observed [13, 15, 19] that sellers with highly positive reputations tend to sell goods at a higher price when compared to sellers with lower reputations, demonstrating the central role that user reputations play in online marketplaces. Malicious buyers (who do not pay for goods purchased) and malicious sellers (who do not deliver the promised goods) quickly gain bad reputations and are avoided [11].

One challenge, however, is that accounts on online marketplaces are often free to create (usually only requiring filling out a form and solving a CAPTCHA [23]), to avoid discouraging potential users. As a result, reputations derived from user feedback are still subject to three types of manipulation:

- Malicious users whose accounts have a bad reputation can effectively *white-wash* their reputation by creating a new account with a blank reputation.
- Malicious users can *collude* by providing positive feedback on each other's transactions, thereby improving both of their reputations.[1]
- Malicious users can create fake identities, known as *Sybils* [7], and use these to provide positive feedback on fictitious transactions between the various identities, thereby inflating their reputations.

Reputation manipulation can lead to significant monetary losses for defrauded users. For example, a single malicious eBay user was recently found to have created 260 different accounts, fabricated positive feedback, and stolen over $717,000 from over 5,000 users [24]. This

[1] In fact, this type of abuse can be plainly viewed on eBay by searching for auctions that are selling "positive feedback." As of this writing, 350 such auctions exist for prices ranging from $0.01 to $0.99.

case is hardly unique: Another malicious eBay user was arrested after defrauding others of over $1 million [20].

In this paper, we propose Bazaar, a system that strengthens user reputations in online marketplaces in the face of collusion, white-washing, and Sybil attacks. Bazaar creates and maintains a *risk network* in order to predict whether potential transactions are likely to be fraudulent. The risk network consists of weighted links between pairs of users who have successfully conducted transactions in the past. When a transaction is about to be completed, Bazaar calculates the max-flow between the buyer and seller; if it is lower than the amount of the transaction, the transaction is flagged as potentially fraudulent. Since Bazaar only needs to determine whether the max-flow is above a given value (instead of calculating the exact max-flow), Bazaar stores the risk network using a novel *multi-graph* representation. We demonstrate that this results in a substantial speed-up of Bazaar's max-flow calculation while imposing only a modest storage overhead.

Bazaar provides a number of useful security properties: First, malicious users in Bazaar cannot conduct more fraud together than they could separately, and as a result, there is no incentive for malicious users to collude. Second, malicious users cannot gain any advantage from conducting Sybil attacks, and thus, there is no incentive to create multiple identities. Third, Bazaar explicitly allows users to create as many identities as they wish; this is sometimes a desired feature in online marketplaces, where sellers may own multiple businesses or wish to maintain separate identities for different types of goods. Fourth, Bazaar provides a strict guarantee that each user can only defraud others by up to the amount of valid transactions the user has participated in, regardless of the number of identities the user possesses, thereby bounding the potential damage.

We evaluate Bazaar using a trace collected from eBay, the largest online marketplace. We collected a 90-day history of five of the most popular categories on the eBay United Kingdom site, encompassing over 3 million users and 8 million auctions. Simulating Bazaar on this data set, we demonstrate that Bazaar successfully bounds the amount of fraudulent transactions that malicious users can conduct, while only rarely impacting the transactions that occur between non-malicious users. We demonstrate that if Bazaar had been deployed on eBay during the 90-day period and in the five categories we study, it would have flagged over £164,000 of auctions that eventually resulted in negative feedback as potentially fraudulent, substantially increasing the reliability of the online marketplace.

The rest of this paper is organized as follows. Section 2 describes the approaches that are currently taken to secure online marketplaces, and Section 3 provides more detail on different types of fraud that are still present today. Section 4 describes the design of Bazaar in detail, and Section 5 details the multi-graph representation of the risk network. Section 6 presents an evaluation of Bazaar. Section 7 details related work and Section 8 concludes.

2 Background

Online marketplaces often use site-specific mechanisms for fraud prevention, but many of these can be reduced to a few simple techniques:

Making joining the market difficult Certain marketplaces only allow trusted users or organizations to participate as sellers, often requiring upfront fees or accounts backed by difficult-to-forge financial information. An example of such such an approach is Amazon Merchants [3], which requires bank account information, a $40-per-month fee, and pre-approval for listing high-fraud-risk goods. However, by making it more difficult to join, this approach reduces the usefulness of the marketplace and severely restricts the population of sellers.

Using a trusted broker In some marketplaces, a middleman participates in the transaction and holds payment until the buyer is satisfied with the transaction. For example, on eBay, there are escrow services that hold money for transactions until the buyer has received the good. However, brokers typically charge a fixed fee and a percentage of the sale,[2] increasing the transaction cost and making escrow practical only for expensive goods (representing a small minority of the goods on typical marketplaces).

Requiring in-person transactions Other marketplaces such as Craigslist require buyers and sellers to be within the same geographical area, ensuring that the participants can meet in person to complete a transaction. This approach allows buyers to inspect goods, and sellers to verify payment, before going through with the transaction. However, this approach also severely restricts who is able to buy and sell goods from each other (as the buyer and seller must live close to each other), limiting its usefulness to local marketplaces.

Providing insurance Certain marketplaces offer buyer and seller insurance programs, either by default or for a fee. However, coverage is generally limited to certain geographic regions and the cost of the insurance payouts and program administration results in higher fees for marketplace users. Nevertheless, the information that Bazaar provides can be viewed as an estimate of risk be-

[2]For example, eBay's recommended escrow service charges a minimum of $22 and up to 3% of the transaction cost.

tween two parties, and can therefore be used as an input when choosing the appropriate the insurance premium.

Paying via trusted services Because certain payment methods (e.g., money orders) are difficult to recover, many marketplaces suggest or require that trusted online payment services (e.g., PayPal) be used. Ideally, such services would link accounts to real-world financial information, making the creation of multiple accounts difficult. However, this is not the case: For example, receiving money with a PayPal account only requires an email address (although financial information is required to withdraw funds). Thus, malicious users can receive money with networks of email-backed accounts, and then send that money to the single, "real" account that is able to withdraw money.

Leveraging feedback Finally, many online marketplaces use feedback provided by users who have participated in transactions. For example, eBay's feedback mechanism calculates a score for each user, consisting of the amount of positive feedback minus the amount of negative feedback. Users with highly positive feedback scores are considered to be more trustworthy, and have been observed to sell goods for higher prices [13, 15, 19]. This approach has the advantage of not restricting marketplace membership and allowing any buyer and seller to participate in a transaction. However, as we will observe in the next section, using feedback is often subject to manipulation by malicious users.

Ideally, we would like to prevent fraud without unnecessarily restricting participation in the online marketplace. The first four approaches above artificially restrict the marketplace by making it either harder to join, more expensive to use, segmenting it based on geography, or spreading the cost of fraud to all users. Thus, we focus on the last approach, leveraging feedback, for the design of Bazaar and present a design that is not subject to the manipulation of existing approaches. Focusing on user feedback also has the advantage that is the mechanism used by the largest online marketplaces, such as eBay, meaning Bazaar could be directly applied to such sites.

3 Examples of malicious behavior

We motivate the design of Bazaar by examining several types of fraud that have been observed in online marketplaces today. The eBay dataset that we use for illustration is fully described in Section 6, however, our purpose here is simply to provide a few motivating examples. In this section, we focus on *malicious sellers* who attempt to defraud buyers, as sellers are largely protected from *malicious buyers* by being allowed to verify payment before shipping the good. To define the fraud we observe, we look at various sellers' feedback history, consisting of entries recording whether the buyer was satisfied with the transaction.

For clarity, we begin by examining the feedback history of a typical seller, shown in Figure 1 (a). Even though over 99% of the seller's feedback is positive, a few items of negative feedback can be observed. A certain low level of negative feedback is expected even for non-malicious sellers, as some buyers may have been unsatisfied with their purchase (e.g., due to the good being lost or damaged in transit, a miscommunication between the participants, or buyer's remorse). We will use similar timeline diagrams throughout the rest of this section.

3.1 Leaving the marketplace

One of the most common types of fraud occurs when a seller participates in the marketplace as a non-malicious user for a period of time, and then turns malicious (often by starting to conduct transactions without ever shipping the goods). As a result, the unsuspecting buyers who have not yet received their goods are defrauded. This type of fraud can be detected once the buyers begin to provide negative feedback, serving as a warning to others. However, malicious users often take advantage of the "window of opportunity" before the negative feedback appears: They can advertise and accept payment for a large number of goods before any user realizes that a fraud has occurred.

An example of such a malicious seller is shown in Figure 1 (b). Towards the end of the seller's timeline, he lists a significant number of goods that are never delivered and eventually result in negative feedback. In fact, this user made significantly more money in aggregate from the fraudulent transactions than from the non-fraudulent transactions. The underlying problem is that *in-progress transactions are not counted against a seller's reputation*, enabling malicious users to establish a reputation, defraud users with the window of opportunity, and then re-join the site with a new account.

3.2 Hiding fraud in the noise

As an alternative to leaving the marketplace, malicious users have also been observed to "hide the fraud in the noise" by participating in many non-fraudulent transactions, but conducting fraudulent transactions for (relatively) expensive goods. As a result, their feedback history has only a small amount of negative feedback, and only a close inspection of the transaction values reveals the fraud. An example of a malicious user conducting such fraud is shown in Figure 1 (c), where the user made more money through the two fraudulent transactions than through the hundreds of non-fraudulent

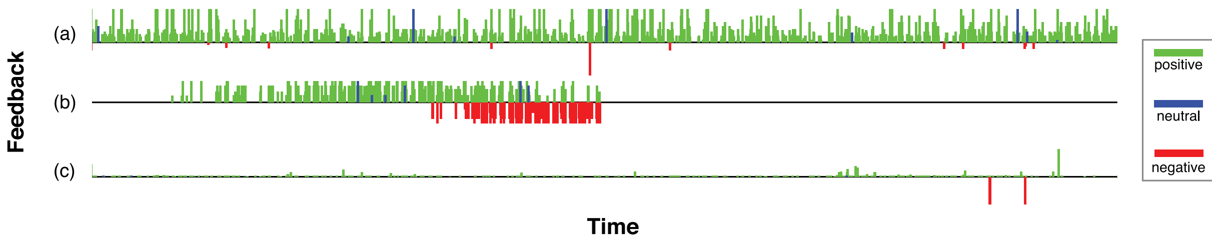


Figure 1: Auction feedback history over time for three eBay sellers: (a) a typical seller, (b) a malicious seller who leaves the marketplace, and (c) a malicious seller who hides the fraud in the noise by conducting a few, large fraudulent transactions. Positive feedback is shown in green, neutral feedback in blue, and negative feedback in red and below the line. The size of each bar correspond to the log of the value of the auction.

transactions. The underlying problem is that *the value of transactions is not considered when determining a seller's reputation*, enabling malicious users to conduct a high-value fraudulent transactions with the same effective penalty (one piece of negative feedback) as a low-value fraudulent transaction.

3.3 Conducting fictitious transactions

Malicious users have also been observed to conduct fictitious transactions and provide fictitious positive feedback. The ultimate goal of these transactions is not to sell a good, but rather, to improve the user's feedback score, making the user look more like an non-malicious user. For example, numerous auctions on eBay are labeled with "Positive Feedback Guaranteed." Often, these auctions ostensibly offer a copy of a digital picture or other token item, so as to appear as a legitimate auction.

Thus, it is easy for a malicious user to arbitrarily manipulate his feedback score by adding spurious positive feedback, so as to appear as a legitimate seller. The underlying problem is that *feedback counts the same, regardless of the other user providing the feedback*. This allows malicious users to conspire to inflate each other's feedback score (or, a single malicious user to do the same via a Sybil attack).

3.4 Summary

In this section, we described three of the most common types of reputation manipulation that are present in the online marketplaces of today. In the next section, we describe the design of Bazaar, which addresses each type of manipulation by (a) considering outstanding transactions, (b) taking into account the value of transactions with positive and negative feedback, and (c) discriminating between different users' feedback, in order to prevent malicious users from artificially inflating their reputation.

4 Bazaar design

We now describe the design of Bazaar.

4.1 Overview

Bazaar is intended to augment an online marketplace, run by a marketplace operator, where buyers and sellers may have no previous relationship and accounts are free to obtain. In such systems, buyers must rely on the reputation of the sellers, represented by feedback from other buyers, to distinguish between non-malicious and malicious users. Thus, the goal of Bazaar is to protect buyers from malicious sellers who manipulate their reputation so as to appear non-malicious. Additionally, we aim to keep the existing model and basic user operations, while significantly reducing the vulnerability to fraud. By doing so, Bazaar serves as a drop-in component applicable to numerous marketplaces.

Now, let us introduce a few definitions that we use for the remainder of this section. A *user* corresponds to an actual person in the offline world. An *identity* is an online account with a particular username associated with it. A user can have a potentially arbitrary number of identities. A transaction is an event where two identities agree to a sale, which has some value. Note that both identities in a transaction may correspond to the same user.

Bazaar relies on two insights. First, successful transactions between different users require significant effort and risk for both parties. Both users are trusting the other to complete the transaction, by providing payment or delivering the good. We refer to this as *shared risk* between two users. Second, once a transaction has been successfully completed, the two users are more likely to enter into a transaction together in the future. Note, however, this risk in not unbounded, and is dependent on the type of transaction that has occurred: The amount of risk that two users are willing to undertake is likely proportional to the amount of risk that has been successfully rewarded.

4.2 Risk network

We view a successful transaction as linking two identities in an undirected fashion, where the weight of the link is the aggregate monetary value of all successful transactions—successfully rewarded shared risk—between the two identities. For example, if identities A and B participated in two successful transactions for $5 and $10, there would be an $A \leftrightarrow B$ link with weight $15. Note that link weights must always be non-negative.

The set of all such links forms an undirected network, which we refer to as the *risk network*. An example of such a network is shown in Figure 2 (a). Note that the risk network has a particularly useful property: The weights are automatically generated by user actions, and do not have to be explicitly provided by users. As we demonstrate below, the risk network can be used not only to gauge the risk between two identities who have conducted a transaction in the past, but also between arbitrary identities who may not have directly interacted in the past.

4.3 Design

Bazaar is run behind-the-scenes by the online marketplace operator. The basic operation of Bazaar is simple: When a buyer is about to enter into a transaction, the marketplace operator queries Bazaar, which calculates the max-flow in the risk network between the buyer and the seller. If the max-flow is below the amount of the potential transaction, the marketplace operator flags the transaction as potentially fraudulent. We discuss ways in which this output can be used by the marketplace operator in Section 4.5, but for now, we assume that flagged transactions are blocked.

The intuition for this approach lies in the observation above about shared risk. Consider a risk network with only two identities, connected by a link of weight w. The identities may be willing to engage in another transaction of value w, and if that is successful, then another transaction for a higher amount. Bazaar generalizes this intuition, allowing identities who are not directly connected to engage in a transaction as long as there is a set of paths of sufficient weight connecting them. For example, in the network shown in Figure 2 (a), if A was about to buy a good from D, Bazaar would consider the flow on paths $A \leftrightarrow B \leftrightarrow D$ and $A \leftrightarrow C \leftrightarrow D$ in order to determine D's reputation from A's perspective.

In existing online marketplaces, feedback-based reputations are "global," in the sense that everyone has the same view of a given user's reputation. In Bazaar, reputations are a function of both the user who is being asked about as well as the user who is asking. As we demonstrate below, this approach allows Bazaar to mitigate reputation manipulation: Malicious users who conspire to inflate their reputations do not necessarily increase their reputations from the perspective of non-malicious users.

4.3.1 Putting credit "on hold"

The design of Bazaar is complicated by the fact that the buyer may not be able to determine whether the transaction was fraudulent immediately after sending payment for the good; generally, there is a delay between when he agrees to the transaction and when the good arrives. In order to prevent malicious sellers from abusing these outstanding transactions in the manner observed in Section 3.1, when the buyer decides to go through with the transaction, Bazaar first determines a path set[3] between the buyer and seller that has a total weight of at least the transaction amount. Such a path set must exist, as, otherwise, the max-flow between the buyer and seller is lower than the transaction amount (meaning Bazaar would have flagged the transaction as potentially fraudulent).

Once the path set is determined, Bazaar temporarily lowers the weights on these paths (in aggregate) by the transaction amount. In essence, this puts the weight on these paths "on hold" until feedback concerning the success or failure of the transaction is received. Since each link weight must always be non-negative, this approach prevents the malicious users from leveraging the weight that is "on hold" in order to conduct additional transactions.

Continuing with our running example in Figure 2, the initial state of the risk network is shown in Figure 2 (a), with each identity having participated in transactions with two other identities. Then, suppose that A conducts a $10 transaction with D. Bazaar determines that the max-flow between A and D is greater than $10, and therefore allows the transaction to go through without being flagged. In doing so, Bazaar temporarily lowers the links along the path set by a total of $10 (specifically, $2 is lowered off of the $A \leftrightarrow B \leftrightarrow D$ path and $8 is lowered off of the $A \leftrightarrow C \leftrightarrow D$ path). This is shown in Figure 2 (b).

4.3.2 Responding to feedback

Finally, once the buyer provides feedback about the transaction, Bazaar makes changes to the risk network. These changes depend on the feedback from the buyer:

- **Positive feedback** If the buyer reports a successful transaction, indicated by positive feedback, Bazaar restores the temporarily lowered weight and additionally creates a new link directly between

[3] If multiple path sets exist that have sufficient weight, Bazaar simply picks one of these sets randomly.

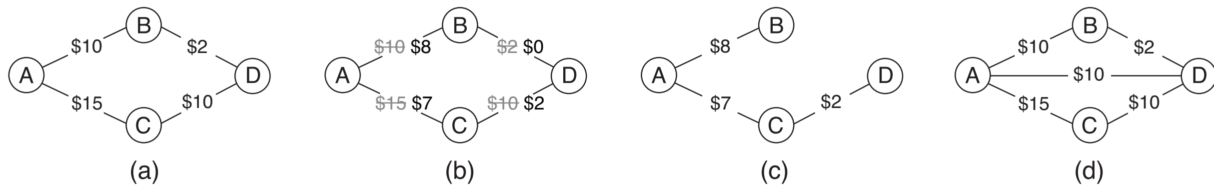


Figure 2: State of the risk network while A conducts a $10 transaction with D. The state is shown (a) before the transaction, (b) while waiting for feedback, (c) if the buyer reports an negative feedback, (d) if the buyer reports a positive feedback, and (a) again, if the buyer reports neutral feedback or the timeout expires.

the buyer and seller weighted by the transaction amount.[4] This has the effect of both restoring the network to its previous state, and creating a new risk link between the buyer and seller. The intuition for this action follows from the discussion above, whereby the buyer and seller are more likely to enter into a future transaction together.

- **Neutral feedback** If the buyer reports a partially successful transaction, indicated by neutral feedback, Bazaar restores the temporarily lowered weight, but does not create a new link. This has the effect of restoring the network to its previous state, with no changes. The intuition for this action is that users who provide neutral feedback are not claiming that the transaction was fraudulent, but are not completely satisfied. Thus, the buyer is not likely to enter into a future transaction with the seller, but does not wish to punish the seller by providing negative feedback.

- **Negative feedback** If the buyer reports an unsuccessful transaction, indicated by negative feedback, Bazaar makes the temporary lowering of the weights permanent and does not create any new links. This has the effect of reducing weight on the seller's links, thereby decreasing the seller's ability to conduct transactions in the future without having them flagged. In particular, if the seller conducts many transactions that end up with negative feedback, eventually, all of his links will be exhausted, and he will be unable to conduct any non-flagged transactions.

- **No feedback** Finally, if the buyer does not report feedback at all, a configurable timeout of T is used, after which Bazaar responds as if the buyer provided neutral feedback (i.e., the temporarily lowered weight is restored, but no new link is created). This is similar to existing sites, which often have a time cutoff for providing feedback.

[4]If a direct link already existed, then Bazaar simply increases that link's weight by the transaction amount.

Returning to our running example in Figure 2, suppose that the feedback is received or the timeout occurs. Bazaar either makes the weight reductions permanent if the buyer reports negative feedback (Figure 2 (c)), restores the previous weights and also forms a new $A \leftrightarrow D$ link if the buyer reports positive feedback (Figure 2 (d)), or restores the previous weights if the buyer reports neutral feedback or the timeout occurs (Figure 2 (a)).

The intuition for why Bazaar is able to prevent fraud is demonstrated by the network shown in Figure 3, where a malicious user X has created a number of identities (X_1 ... X_5) and has conducted fictitious transactions between them (in essence, the weight on these links can be arbitrarily set by X). Without Bazaar, potential victim Z would only see X_1's fictitious feedback consisting of a number of positive entries. Not knowing that all of this positive feedback was from other identities owned by the same underlying user, Z would likely be defrauded. With Bazaar, however, the fictitious transactions do not contribute to the max-flow between Z and X_1, and Bazaar is likely to flag the transaction as potentially fraudulent (even though Bazaar had no a priori knowledge that all X_i identities belong to the same user). Moreover, should

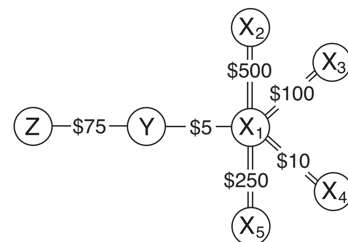


Figure 3: Example risk network, showing why Bazaar secures reputations (links represent previous real transactions, and double links represent fictitious transactions). Honest identity Z is considering entering into a transaction with malicious identity X_1 (owned by the same user as X_2 ... X_5). Without Bazaar, X_1 appears to be a reputable seller. With Bazaar, the fictitious transactions do not increase the max-flow ($5) between Z and X_1, thereby preventing the reputation manipulation.

X use one of these identities to conduct a fraud—of no more than \$5, since anything greater would be automatically flagged as potentially fraudulent—the $Y \leftrightarrow X_1$ link will have credit put "on hold" and eventually reduced (once the buyer provides negative feedback), regardless of which identity X selects as the seller. This is the case regardless of the number of identities X creates or how he creates fictitious transactions between them. In effect, Bazaar forces X to participate in successful transactions with other non-malicious users in order to increase his max-flow, and penalizes these links whenever X conducts fraud.

4.3.3 Bootstrapping

New users, by definition, have no transaction history and therefore have a max-flow of 0 to all other users. To allow new users to participate without having all of their transactions flagged as potentially fraudulent, Bazaar uses two techniques. First, Bazaar allows users to create virtual links to their real-world friends (in the same manner as malicious users can create links in the risk network between their identities by conducting fictitious transactions). This mechanism allows users to obtain a few "starter" links from the friends, without opening a new security vulnerability: Since the user's friends are, in effect, vouching for the new user, the friends are putting their existing links on-the-line. If the new user defrauds others, not only would his links be penalized, but the links of his friends would be as well.

Second, if the new user does not have any real-world friends in the marketplace, Bazaar allows him to optionally provide the marketplace operator with an amount of money to hold in escrow. In return, the marketplace operator creates links between the new user's identity and other, random identities with a total value of the amount in escrow. These newly created links allow the new user to participate in the marketplace. At some later time, the new user can request that the escrowed money be returned (and the marketplace operator will remove the created links). However, if the created links represent weight on hold, or if the they have been lost (due to a fraudulent transaction), the marketplace operator would refuse to return the escrowed money. This approach does not open up a new vector for attack, as (a) the most the new user could defraud is the amount of escrowed money, and (b) if the user does commit such a fraud, he would lose his escrowed money. In essence, such an attack would not allow a malicious user to gain any money.

4.4 Guarantees

We now discuss the guarantees that Bazaar provides. In brief, Bazaar ensures that malicious users can only defraud others up to the total amount of successful transactions that they have participated in with non-malicious users. To see this, let us imagine a malicious user X, whose identity has outgoing links with weight totaling a_X. Each time X conducts a fraudulent transaction, some of his links are reduced, in aggregate, by the amount that he defrauds. Thus, once X has defrauded a total of a_X, all of his links have been removed and he is prevented from participating in transactions in the future. Moreover, X cannot use the "window of opportunity" (discussed in Section 3) to conduct fraud before feedback is provided, as Bazaar puts link weights on hold until the feedback is received.

Moreover, the same analysis holds for any subgraph or any cut in the network. Thus, collusion between malicious users does not help; the users can only defraud together for the total of what they could defraud separately. This argument also explains why creating fake identities also does not help, as it is the cut in the network between the user's identities and the rest of the network that bounds the amount that the user can defraud, instead of the number of identities the user has or the amount of fictitious feedback. The upshot is that Bazaar does not explicitly detect Sybil nodes or malicious users in the network, rather, it provides a strict guarantee on the amount of fraud that they are able to conduct.

The implication of this analysis is that we can characterize the amount of fraud the malicious users are able to conduct, in aggregate. Let us partition the network in two groups: G, containing non-malicious identities who do not conduct fraudulent transactions, and M, containing malicious identities whose goal is to defraud others. Let us consider the cut in the network between these two sets, with total value c_{MG}. We make two observations: First, any links that lie along this cut must represent non-fraudulent transactions between non-malicious users and malicious users; in essence, these represent instances where the malicious users were non-malicious. Second, any time one of the malicious users defrauds a non-malicious user, this cut is reduced by the amount of the fraud. Thus, malicious users can only defraud non-malicious users of up to c_{MG} before the two groups are partitioned and all of the malicious users' transactions are flagged as potentially fraudulent to the non-malicious users.

It is worth noting that this is a much stronger guarantee than what can be provided today. For example, today, a user can potentially purchase a large amount of fictitious positive feedback with a low monetary investment, use that feedback to appear as an non-malicious seller, and then defraud users of a significant amount of money. This problem is exacerbated by the fact that the defrauded users have to realize that they have been defrauded before they can provide negative feedback and

warn others, leaving a significant window of vulnerability. Moreover, the malicious user can simply repeat this process with a new identity. By putting this bound in place, we are able to force the malicious user to participate in valid transactions with non-malicious users, thereby significantly reducing the attractiveness of committing such a fraud.

4.5 Discussion

We now discuss a few deployment issues with Bazaar.

User interaction The marketplace operator can use the output of Bazaar in multiple ways. For example, the marketplace operator can provide strong fraud guarantees by not allowing flagged transactions to go through. Alternatively, the marketplace operator can require that flagged transactions use an escrow service or insurance service, or can more closely scrutinize the transaction. The latter options represent an additional incentive for the marketplace operator to deploy Bazaar, as selling additional services such as escrow or insurance may increase their revenue while at the same time attracting customers due to a decrease in fraud.

Providing honest feedback An additional concern is whether buyers are incentivized to provide honest feedback on transactions in Bazaar. First, rational buyers have no incentive to provide incorrect negative feedback: By doing so, they penalize their own links and they prevent the creation of a new link between themselves and the seller. Since having more links is desirable (as it allows a user to participate in more and higher-valued transactions), buyers are disincentivized from providing incorrect negative feedback. Second, rational buyers also have no incentive to provide incorrect positive feedback. In particular, if they were unhappy with the transaction, providing positive feedback creates a new direct link to the seller; this is likely to be highly undesirable if the buyer felt defrauded, as it risks the buyer's existing links.

Targeted attacks Another possible concern is whether Bazaar introduces a new attack vector by allowing a malicious user to conduct a targeted attack on a seller by purchasing their goods and then always providing negative feedback (thereby damaging the seller's reputation). First, such an attack is possible in existing marketplaces, as malicious users can conduct this attack by creating numerous free identities and then purchasing the victim's goods. Thus, Bazaar does not open up a new avenue for attack. Second, we note that Bazaar raises the bar on this attack, making it more difficult to conduct: With today's marketplaces, the malicious users can purchase the victim's goods immediately after creating another identity. With Bazaar, the malicious users must first conduct non-fraudulent transactions in order to obtain enough links to be able to conduct the attack, making such an attack significantly more difficult and less attractive.

Compromised accounts If a user's account password is compromised, an attacker can conduct fraudulent transactions on the user's behalf, eventually causing the user to run out of links. However, this attack is not unique to Bazaar, since attackers could conduct the same attack with the reputation systems in-use today. Moreover, with Bazaar, the amount of fraud that can be conducted is still subject to the Bazaar bounds, whereas without Bazaar, it is potentially unbounded.

Protecting sellers Bazaar, as described so far, focuses on protecting buyers from being defrauded by malicious sellers who manipulate their reputation. However, in certain marketplaces, it may be necessary to protect sellers as well (e.g., from buyers who use fraudulent payment mechanisms like stolen credit cards). We leave protecting sellers to future work, with one comment: The need to protect sellers is somewhat mitigated by the fact that marketplace operators generally allow sellers to verify payment before shipping the good.

Maintaining full network knowledge The design of Bazaar proposed so far requires knowledge of the complete risk network. This is not an unreasonable assumption, as online marketplaces are generally run by a single operator that has full knowledge of all transactions. Given this information, the marketplace operator can create and update the risk network as necessary. It may be possible to decentralize knowledge of the risk network, but this remains an open research question and is a subject of future work. A decentralized system has several advantages with regards to privacy and scalability, but as we do not know of any decentralized online marketplaces, the path to deploy a decentralized solution is unclear.

5 Calculating max-flow using multi-graphs

The Bazaar design described so far relies on finding the max-flow path between two nodes in order to calculate the amount of risk embedded in a potential transaction. Since the risk network may have large number nodes and links, finding the max-flow between nodes using traditional approaches like Ford-Fulkerson [8] and Goldberg-Rao [9] may prove to be expensive. Similarly, pre-computing max-flow values through techniques like Gomory-Hu Trees [12] may also prove too costly, and are complicated by the fact that the risk network is changing over time. Instead, Bazaar uses a novel approach called *multi-graphs* in order to reduce the computation required. In this section, we first describe useful observations on risk networks and of our desired max-flow al-

gorithm, detail the multi-graph data structure, and finally demonstrate how multi-graphs reduce the complexity of finding max-flow values.

5.1 Observations

We begin by making two observations concerning the risk networks in online marketplaces and the properties of the max-flow calculation in Bazaar.

1. **Dense core** First, like social networks [16], the risk networks we observe in real-world online marketplaces tend to have a dense core, meaning a small minority of users possess the majority of the links. Moreover, the higher-valued links (representing risk relationships with higher values) also tend to fall in this "core." As a result, the risk network tends to shrink rapidly if links with less than a specified weight are discarded. We demonstrate this with real-world data in the following section.

2. **Actual max-flow not needed** Second, and most important, Bazaar does not need to actually calculate the value of the max-flow between a potential buyer and seller. Instead, Bazaar simply needs to verify whether the max-flow is above a certain value (i.e., the value of the potential transaction). This implies that the complexity of calculating the max-flow in Bazaar may not be as high as a general max-flow calculation.

The multi-graph optimization, described next, leverages both of these observations in order to reduce the complexity of the max-flow calculation in Bazaar.

5.2 Multi-graphs

Formally, we define a multi-graph M to be a set of graphs

$$M = \{G_0, G_1, ..., G_n\}$$

where each graph $G_i = (V_i, E_i)$. These graphs are related: First, G_0 is defined to be the entire risk network. Second, G_i is defined to be the subgraph of G_{i-1} with

$$\begin{aligned} E_i &= \{e \in E_{i-1} : w(e) \geq k^i\} \\ V_i &= \{v : (v, \cdot) \in E_i\} \end{aligned}$$

where $w(e)$ represents the weight of edge e and k is a configurable system parameter with a suggested value of 2. Thus, the multi-graph contains a series of risk networks, where each subsequent network is a subgraph of the previous containing only those links with an exponentially higher weight. An example of converting a risk network into a multi-graph is shown in Figure 4.

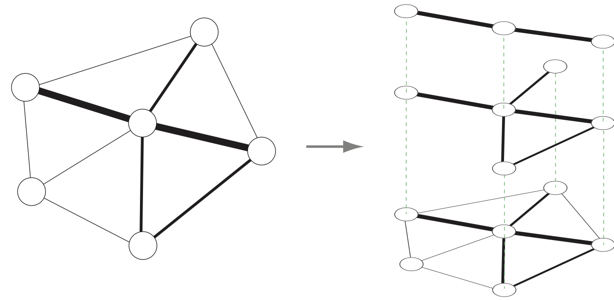


Figure 4: Conversion of a risk network (left) to a risk multi-graph (right). Links with higher weights are shown with thicker lines. Graphs at higher levels in the multi-graph only include links with exponentially increasing weights (e.g., with $k = 2$, the three levels of the multi-graph would represent all links, links with weight \$2 and higher, and links with weight \$4 and higher).

Note that a multi-graph contains multiple copies of a given link, the weights of which need to be kept consistent. There are three operations on the risk network under which Bazaar must maintain consistency:

- **Link addition** When a new link is added, it is simply added to all of the graphs to which it belongs (e.g., if the link weight is w, the link is added to $\{G_i : w \geq k^i\}$).
- **Link weight change** When the weight of a link is changed, it is simply added to or removed from the appropriate graphs. Conceptually, this can be viewed as removing the link from all graphs, followed by adding it back at its new value.
- **Link weight temporary adjustment** Recall that Bazaar may temporarily lower the weight of a link when a transaction is in progress. Conceptually, this can be viewed as changing the weight of the link. Later, if the adjustment is undone, this can again be viewed as a weight change.

5.3 Max-flow on multi-graphs

Now, let us consider what happens when Bazaar calculates whether a path set of total weight w exists between a source and destination. With a normal risk network, Bazaar must use an algorithm like Goldberg-Rao, which runs over the entire risk network and is optimized to determine the actual max-flow between the source and destination.In contrast, with a multi-graph, Bazaar proceeds by first finding the highest-weight network G_m where both the source and the destination are present. Then, Bazaar runs any existing max-flow algorithm on G_m, looking for a set of paths of collective weight w. If such a

set is found, then the algorithm returns that set and is finished. If no such set is found, Bazaar repeats the process with the next-lowest graph G_{m-1}. This process continues until either a set of paths of weight w is found, or Bazaar cannot find such a set of paths in the lowest graph G_0. The latter case indicates that the max-flow in the original risk network was lower than w, demonstrating that finding the max-flow in a multi-graph is guaranteed to have the same outcome as finding the max-flow in the original risk network.

It is worth noting that multi-graphs require an increase in storage costs, since multiple copies of many links must be stored. However, as we demonstrate in the evaluation, the storage requirements of the multi-graphs are modest and are easily met by today's computing hardware.

5.4 Benefit of multi-graphs

We now describe how the use of multi-graphs speeds up the max-flow calculation in Bazaar. Consider the case of a transaction of value w. First, because of observation 1 above, the sizes of the graphs G_i decrease extremely rapidly as i increases. Thus, running a max-flow algorithm over G_i is significantly faster than running it over G_{i-1}. Second, because of observation 2, it is possible to modify the max-flow algorithm to terminate as soon as it finds a path set of weight w, instead of continuing to find the actual max-flow. For example, if we are using Ford-Fulkerson, only a few rounds may be are needed in order to find a set of paths of weight w. Third, the increasing link weights in higher G_i further reduce the running time of the max-flow algorithm, as the path set in higher G_i is likely to consist of only a few paths. As we demonstrate in the evaluation, these effects allow multi-graphs to significantly speed up the calculation in practice.

6 Evaluation

In this section, we present an evaluation of Bazaar. In particular, we use data collected from a real-world online marketplace to determine if the max-flow technique employed by Bazaar is able to detect and prevent fraudulent transactions. We describe the data collected, verify our observations in the previous section, demonstrate the performance gains of using multi-graphs, and present an evaluation of Bazaar on real-world data.

6.1 Auction data

In order to evaluate Bazaar, we collect data from eBay, the largest online marketplace. We focus on collecting data from the `ebay.co.uk` site, containing United Kingdom auctions.

Category	Purchases	Users	Avg. Price
Clothes	3,311,878	1,436,059	£9.73
Collectibles	940,815	454,773	8.90
Computing	964,925	661,285	21.31
Electronics	861,108	652,350	20.67
Home/Garden	2,795,795	1,426,785	16.57
Total	8,874,521	3,168,455	£14.12

Table 1: Distribution and monetary values of feedback seen in our trace.

eBay makes the feedback for all users public. Each piece of feedback consists of the feedback value (positive, negative, or neutral), the auction the feedback was for, the identity of the user providing feedback, and a short message from that user explaining the feedback. Feedback can be provided by both the buyer and seller, so each auction can result in two pieces of feedback. eBay only makes detailed feedback available for 90 days, after which time, information about the auction the feedback is for is removed, and only the feedback value, message, and providing user remain. Thus, we are only able to collect detailed feedback for the previous 90 days.

eBay provides an API to collect data, but rate limits the requests to a very low rate. Instead, we use web scraping to collect data. We start from one user and crawl their feedback profile. From this profile, we learn about other users and proceed to crawl them. We continue this process until we exhaust all known users, effectively performing a breadth-first-search of the feedback graph.

In order to make our data collection process tractable, we only consider auctions and feedback that occur in five of the largest auction categories, shown in Table 1. Thus, we do not crawl other users that appear in the feedback history if the auction is not in one of these five categories. Since eBay allows users to participate in international transactions, not all users we discover are located in the United Kingdom. We restrict our crawl to only consider users located in United Kingdom, leaving us with a total of 3,168,455 distinct users (note that users may participate in multiple categories). Finally, because Bazaar focuses on protecting buyers from malicious sellers, we only collect feedback from buyers to sellers (and ignore feedback from sellers to buyers). In total, our dataset contains information on 8,874,521 items of feedback.

6.2 Dense core of risk networks

We now turn to validate our observation in Section 5 that motived our multi-graph design. Specifically, we examine whether there tends to be a dense "core" of users in the risk network, which was necessary for the multi-graph representation to have acceptable overhead. To do so, we use a similar approach to prior studies [16] and ex-

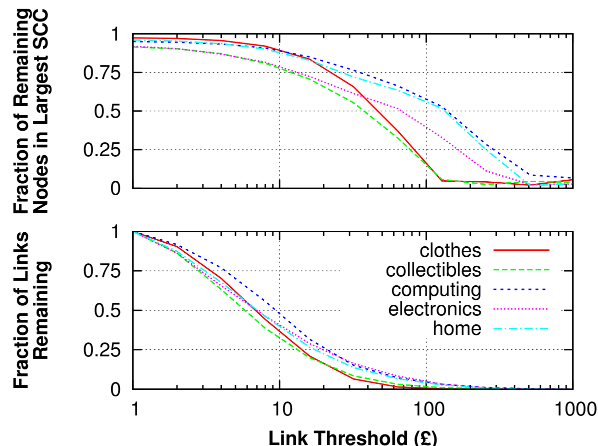


Figure 5: Fraction of links remaining (bottom) and fraction of the remaining nodes in the largest SCC (top) as only higher-weighted links are considered. Even as the majority of links are discarded, the largest SCC still contains most nodes, indicating the presence of a core.

amine the subgraph consisting of highly weighted links. We are interested in both the size and the connectedness of these subgraphs. Figure 5 shows how these two attributes vary as only higher-weighted links are considered. As the threshold rises from £1 to £20, almost 80% of the links are discarded. However, the vast majority of the remaining nodes are still in the largest strongly connected component (SCC), indicating the presence of a strong core. For some of the categories, the largest SCC does not disintegrate until only links of over £100 are considered. This validates our observation from the previous section, and indicates that multi-graphs are likely to speed up Bazaar's max-flow calculations in practice.

6.3 Multi-graph performance

We now turn to evaluate the benefits of using the multi-graph representation on the performance of finding max-flow paths. Specifically, we examine the tradeoff between memory and speed; since multi-graphs store multiple copies of certain links, they naturally have higher memory requirements than only using a risk network. First, we show the number of multi-graph levels and the resulting memory overhead, relative to the single graph, of storing a multi-graph in Bazaar in Table 2. As can be seen from the table, while the relative storage overhead is a 3- to 4-fold, the absolute overhead is small.

Next, we turn to evaluate the speedup of verifying whether a max-flow exists using a multi-graph in Bazaar. To do so, we create separate risk networks from each of the five categories by aggregating our feedback trace, creating links between users who participated in transactions with positive feedback. We then randomly select

Category	Size (MB)	Levels	Overhead Rel.	Overhead Abs. (MB)
Clothes	7.38	12	234.6%	17.3
Collectibles	2.01	14	221.0%	4.44
Computing	3.47	13	282.9%	9.83
Electronics	3.23	13	255.9%	8.25
Home/Garden	7.31	13	251.8%	18.4

Table 2: Memory requirements of a single graph representation of the risk network, and number of levels and overhead (both relative and absolute) of a multi-graph representation, with $k = 2$.

1,000 pairs of nodes from each category and an amount from the prices in the observed auction trace. We calculate the time required to verify whether a set of paths exist with at least the selected auction amount between the pair of users. For this experiment, we used a machine with a 2.83 GHz Intel Xeon processor.

Table 3 presents the results of this experiment. Using the multi-graph representation shows a significant performance gain, with speed-ups ranging between 1.92× and 2.86×. In fact, with the multi-graph, most of the max-flow calculations take less than 6 seconds to complete. However, most of the calculations that are successful (e.g., a set of paths is found with at least the specified weight) finish quickly, while the calculations that eventually fail (e.g., no such set is found) take much longer to finish, thereby inflating the average. This trend is expected since a failure must traverse every graph in the multigraph, whereas a success has the potential to end early. This observation suggests a further avenue for speeding up the max-flow calculation in practice, by considering calculations that run longer than a specified amount of time to have failed. For example, in the Computing category, if all calculations that take longer than two seconds are considered to have failed, this would only misclassify 5.5% of the eventually to-succeed calculations, and would lower the average running time from 1.66 to 0.70 seconds.

Regardless, even without this further optimization, the average max-flow calculations in the largest category we examine (Clothes) required 6.29 seconds, meaning that 13,736 calculations could be completed per server per day. Using our trace, we determined that the highest number of auctions closing on a single day in this category was 80,846, meaning that Bazaar could be deployed in this category by purchasing a server with at least 6 cores. Of course, synchronization would need to be maintained to ensure that two cores were not using a single link at once. We observed, though, that such conflicts occur rarely (0.0165% of the time in this category), implying that parallelism of the max-flow algorithm [1] is likely to provide significant performance gains.

Category	Time (s) Single	Multi-graph	Speedup
Clothes	18.0	6.29	2.86×
Collectibles	2.53	1.18	2.14×
Computing	3.78	1.66	2.27×
Electronics	2.71	1.41	1.92×
Home/Garden	11.6	5.34	2.15×

Table 3: Average max-flow calculation times, and relative speedup when using multi-graphs with $k = 2$.

Category	Fraction of transactions incorrectly flagged
Clothes	1.11%
Collectibles	1.12%
Computing	3.23%
Electronics	4.68%
Home/Garden	2.43%

Table 4: Fraction of non-fraudulent transactions that are incorrectly flagged as fraudulent by Bazaar. The fraction flagged incorrect is never higher than 5%, indicating that non-malicious users are largely unaffected.

6.4 Detecting fraud with Bazaar

We now turn to examine how well Bazaar is able to detect fraudulent transactions. In particular, we are interested in three aspects of Bazaar's performance: First, what is the impact on non-malicious users? In other words, how often are non-malicious users' transactions incorrectly flagged as potentially fraudulent? Second, is Bazaar able to bound the amount of fraud that malicious users are able to conduct? Third, what impact, in terms of the amount of fraud prevented, could we expect from Bazaar if it were deployed on a online marketplace?

To conduct the evaluation, we use a random subset of 80% of the feedback data to create a risk network for each of the five categories, and then use the remaining 20% of the feedback data to simulate the operation of Bazaar. Because our data only represents a 90-day period, many of the users participate only in a single transaction (and therefore have a max-flow of 0 to all other users). In order to reduce the bias caused by our short time-window of data, we only simulate users who we observe to participate in at least five transactions during the time range. Finally, for each data point, we repeat the experiment 10 times using different random seeds.

To simulate Bazaar, we need a few pieces of information from each auction transaction: the identity of the buyer and seller, the price of the auction, the purchase and feedback time, and the feedback itself. Our crawled data unfortunately only contains the purchase time for 54.6% of the data.[5] So, for the auctions where the purchase time is not available, we artificially select a purchase time by subtracting a random "delay" from the feedback time. This delay is randomly drawn from the observed purchase-time-to-feedback-time delay distribution of the other auctions.

6.4.1 Impact on non-malicious users

Our first evaluation examines the potential negative impact that Bazaar has on non-malicious buyers and sellers. The primary form that such impact takes is incorrectly flagging transactions as potentially fraudulent. To determine the frequency with which this happens, we simulate Bazaar without any malicious users and calculate the fraction of transactions that had positive feedback but that would have been flagged by Bazaar due to insufficient max-flow. The results of this experiment are shown in Table 4, listing the fraction of non-fraudulent transactions which are flagged as potentially fraudulent by Bazaar. The results show that no more than 5% of all non-fraudulent transactions are flagged, indicating that non-malicious users in Bazaar are largely unaffected.

6.4.2 Blocking malicious users

We now evaluate whether Bazaar is able to bound the amount of fraud that malicious users can conduct in practice. Recall that Bazaar guarantees that each user is only able to conduct fraudulent transactions up to the amount of non-fraudulent transactions that he has participated in. Thus, we are interested in comparing how much fraud malicious users can conduct, relative to the amount of non-fraudulent transactions they participated in.

To simulate the behavior of malicious users, consistent with prior studies [22], we randomly select 1% of the users to be malicious. For each user, we simulate Bazaar running with other, randomly selected users purchasing items from the malicious user. We then calculate the total amount of fraudulent transactions that each user can conduct, until the point at which Bazaar flags all transactions with the malicious user as potentially fraudulent.

Figure 6 presents the results from conducting this experiment, by plotting the amount of fraudulent transactions a malicious user can conduct versus the sum of the malicious user's initial links. As can clearly be seen in the figure, Bazaar's bound on the amount of fraudulent transactions holds: the amount of possible fraud is strictly bounded by the sum of the non-fraudulent transactions that the malicious user has participated in so far.[6]

[5] In more detail, the purchase time of fixed-price auctions—where a user sells multiple, identical items at a fixed price—is not available, as these auctions have multiple buyers purchasing the items.

[6] A careful reader will note that malicious users are sometimes bounded to less than the actual total of their previous successful transactions. This occurs when, for example, a malicious user is the only

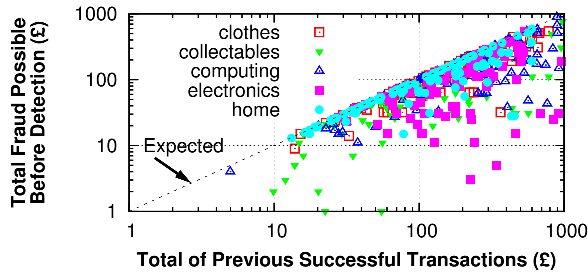


Figure 6: Aggregate amount of fraudulent transactions that malicious users can conduct versus the aggregate value of previous successful transactions. Also included is the expected bound ($y = x$). As expected, Bazaar ensures that malicious users can only commit fraud up the amount of successful transactions that they have participated in previously.

Even if the malicious user whitewashes his account (by creating a new identity), or conducts a Sybil attack (by creating multiple identities and linking them by fictitious transactions), he is unable to conduct any more transactions that are not flagged as potentially fraudulent.

6.4.3 Preventing fraud

As a final point of evaluation, we examine the amount of fraud that Bazaar would prevent, were it to be deployed on a real-world online marketplace. In other words, what impact could we expect from Bazaar?

To evaluate this, we use the same 90-day trace from the five eBay categories. Then, for each seller, we calculate the total amount of goods sold with positive feedback, and the total with negative feedback. Recall that Bazaar prevents any user from having more (price weighted) negative feedback than positive feedback, so the auctions that represent the excess negative feedback would have been flagged as potentially fraudulent. We therefore calculate the total of this excess, and determine what fraction of the overall negative feedback it represents.

Table 5 presents the results. Bazaar would have flagged between 29% and 42% of all auctions that resulted in negative feedback as being potentially fraudulent, thereby possibly preventing these auctions from occurring. While we cannot say that all of these transactions represent fraud (e.g., the negative feedback could simply represent buyer's remorse), the fact that these all come from sellers whose weighted negative feedback is greater than their weighted positive feedback strongly suggests so. In total, the auctions that Bazaar would have prevented represent £164,791.55 worth of goods, significantly bolstering the reliability of the online marketplace. Moreover, this amount is only for a 90-day period in the five categories we study; the amount is likely to be significantly higher if Bazaar were deployed on the entire marketplace and over a longer period of time.

user that another user is linked to: Even though the malicious user's total is increased, this link does not increase the max-flow to any other users (much in the manner of the $X_2 ... X_5$ identities in Figure 3).

Category	Total flagged	Fraction of all negative feedback
Clothes	£28,291.34	29.9%
Collectibles	4,995.04	38.2%
Computing	48,742.66	39.7%
Electronics	34,476.87	42.6%
Home/Garden	47,285.64	32.4%
Total	£164,791.55	36.0%

Table 5: Total number of auctions with negative feedback that would be flagged as potentially fraudulent, and the fraction of all auctions with negative feedback that this represents. Overall, Bazaar would have flagged £164,791.55 worth of auctions that eventually resulted in negative feedback, representing 36% of all such auctions.

7 Related work

Researchers have previously studied approaches to detecting auction fraud, usually relying on machine-learning techniques [4, 18] based on bidding behavior. While these techniques succeed at detecting some fraudulent users, they rely on characteristics of malicious behavior. As a result, unlike Bazaar, these approaches do not provide a bound on the amount of fraud any user can conduct. Additionally, researchers have developed techniques [14, 21] to detect shill bidding, where users conspire with others to artificially inflate the selling price of their auctions. Bazaar is complementary to this work, as it is not concerned with shill bidding, but rather, fraud caused by reputation manipulation.

Other work [5, 10] has examined building reputations based on social relationships between users. While some of the techniques used are similar to Bazaar, Bazaar must determine pairs of trusting users itself (instead of assuming pairwise trust is externally provided). This introduces significant challenges, but enables Bazaar to be deployed on existing sites.

There is also significant work that studies the network formed by users who trust each other, and a number of research systems have already been proposed to leverage this trust. Perhaps the most well known of these are the PGP web of trust [27] and the Advagato trust metric [2]. However, these systems are generally concerned with providing a stronger notion of identity, instead of bounding the amount of malicious activity.

More generally, recent work has focused on detecting Sybil accounts using social networks [6, 25, 26]. These

approaches are not directly applicable to online marketplaces for two reasons: First, they assume the existence of a social network that is not necessarily present, and second, they only bound the number of Sybil accounts that are admitted, not on the amount of fraud that malicious users can conduct. Thus, even with Sybil detection algorithms, malicious users are still able to conspire to arbitrarily inflate each others' reputations.

Like other work [22], Bazaar uses a mechanism that is loosely based on the one used in Ostra [17], a system that uses a social network to block senders of unwanted communication. However, Bazaar differs from Ostra in three important ways. First, while Ostra is based on a relatively stable, unweighted social network, Bazaar uses a weighted risk network that is changing with every transaction (e.g., links are added and removed, and the links weights can grow and shrink over time). Second, Ostra assumes the trust network is given from an external source, while Bazaar constructs the risk network during the operation of the system. This requires Bazaar to face additional challenges, as malicious users are able to create links by participating in transactions (this is not possible in Ostra, as Ostra's assumption is simply that links to non-malicious users take effort to form and maintain). Third, Bazaar works by calculating the max-flow in the risk network, instead of simply finding a single path (as in Ostra). This induces significant engineering challenges and results in a system with a different set of guarantees.

8 Conclusion

In this paper, we presented Bazaar, a system that strengthens user reputations in online marketplaces. Bazaar is based on max-flow calculations over a risk network, a data structure that encodes the amount of rewarded shared risk between participants. Using data on over 8 million purchases from a real-world online marketplace, we demonstrated that Bazaar is able to effectively bound the fraud that malicious users are able to conduct, while only rarely impacting the transactions conducted between non-malicious users.

Given the popularity of online marketplaces and the large amount of fraud that such marketplaces currently experience, our hope is that Bazaar can be used as a drop-in component on real-world sites. Bazaar is designed to be readily applied to such marketplaces.

Acknowledgements

We thank the anonymous reviewers, Peter Druschel, Lakshmi Subramanian, Bimal Viswanath, and our shepherd, Dina Katabi, for their helpful comments. This research was supported in part by NSF grant IIS-0964465 and an Amazon Web Services in Education Grant.

References

[1] R. J. Anderson and J. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. *SPAA*, 1992.

[2] Advagato Trust Metric. `http://www.advogato.org/trust-metric.html`.

[3] Amazon Merchants and Marketplace. `http://www.amazonservices.com/content/sell-on-amazon`.

[4] D. H. Chau, S. P, and C. Faloutsos. Detecting fraudulent personalities in networks of online auctioneers. *PKDD*, 2006.

[5] D. DeFigueiredo and E. T. Barr. TrustDavis: A Non-Exploitable Online Reputation System. *CEC*, 2005.

[6] G. Danezis and P. Mittal. SybilInfer: Detecting Sybil Nodes using Social Networks. *NDSS*, 2009.

[7] J. Douceur. The Sybil Attack. *IPTPS*, 2002.

[8] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Can. J. Math.*, 8, 1956.

[9] A. Goldberg and S. Rao. Flows in Undirected Unit Capacity Networks. *FOCS*, 1997.

[10] A. Ghosh, M. Mahdian, D. Reeves, D. Pennock, and R. Fugger. Mechanism Design on Trust Networks. *WINE*, 2007.

[11] D. G. Gregg and J. E. Scott. The Role of Reputation Systems in Reducing on-Line Auction Fraud. *Int. J. Elec. Comm.*, 10(3), 2006.

[12] R. E. Gomory and T.C. Hu. Multi-Terminal Network Flows. *SIAM*, 9(4), 1961.

[13] D. Houser and J. Wooders. Reputation in Auctions: Theory, and Evidence from eBay. *Econ. Strat.*, 15, 2006.

[14] R. J. Kauffman and C. A. Wood. Running up the bid: Detecting, predicting, and preventing reserve price shilling in online auctions. *ICEC*, 2003.

[15] D. Lucking-Reiley, D. Bryan, N. Prasad, and D. Reeves. Pennies from eBay: The determinants of price in online auctions. *Indus. Econ.*, 55(2), 2007.

[16] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. *IMC*, 2007.

[17] A. Mislove, A. Post, K. P. Gummadi, and P. Druschel. Ostra: Leverging trust to thwart unwanted communication. *NSDI*, 2008.

[18] S. Pandit, D. H. Chau, S. Wang, and C. Faloutsos. Netprobe: a fast and scalable system for fraud detection in online auction networks. *WWW*, 2007.

[19] P. Resnick and R. Zeckhauser. Trust Among Strangers in Internet Transactions: Empirical Analysis of eBay's Reputation System. *The Economics of the Internet and E-Commerce*, volume 11, Elsevier Science, 2002.

[20] B. Sullivan. Man arrested in huge eBay fraud. 2003. `http://www.msnbc.msn.com/id/3078461/`.

[21] H. S. Shah, N. R. Joshi, A. Sureka, and P. R. Wurman. Mining eBay: Bidding strategies and shill detection. *WebKDD*, 2002.

[22] N. Tran, B. Min, J. Li, and L. Subramanian. Sybil-Resilient Online Content Voting. *NSDI*, 2009.

[23] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. *EuroCrypt*, 2003.

[24] J. Weaver. How a bold eBay scam was tracked to South Florida. *The Miami Herald*, 2010.

[25] H. Yu, P. B. Gibbons, M. Kaminsky, and F. Xiao. SybilLimit: A Near-Optimal Social Network Defense Against Sybil Attacks. *IEEE S&P*, 2008.

[26] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. *SIGCOMM*, 2006.

[27] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, 1994.

Dewdrop: An Energy-Aware Runtime for Computational RFID

Michael Buettner*, Ben Greenstein† and David Wetherall*†
University of Washington and Intel Labs Seattle†*

Abstract

Computational RFID (CRFID) tags embed sensing and computation into the physical world. The operation of the tags is limited by the RF energy that can be harvested from a nearby power source. We present a CRFID runtime, *Dewdrop*, that makes effective use of the harvested energy. *Dewdrop* treats iterative tasks as a scheduling problem to balance task demands with available energy, both of which vary over time. It adapts the start time of the next task iteration to consistently run well over a range of distances between tags and a power source, for different numbers of tags in the vicinity, and for light and heavy tasks. We have implemented *Dewdrop* on top of the WISP CRFID tag. Our experiments show that, compared to normal WISP operation, *Dewdrop* doubles the operating range for heavy tasks and significantly increases the task rate for tags receiving the least energy, all without decreasing the rate in other situations. Using offline testing, we find that *Dewdrop* runs tasks at better than 90% of the best rate possible.

1 Introduction

Computational RFID (CRFID) tags are an emerging technology in which sensing and computational abilities are added to traditional RFID tags. Passive UHF RFID tags run and transmit an identifier using energy gathered from the transmissions of nearby RFID readers; they are very small and have no battery or long-term energy store. This ability makes them widely useful in commercial settings to, for example, automate interactions with passports and drivers licenses, identify animals, and track retail goods in manufacturing and supply chains. The addition of sensing and computation with CRFIDs enables a broader range of sensing applications, including cold-chain monitoring, access control, embedded monitoring of bridges and planes, gestural interfaces, activity recognition, and non-intrusive physiological monitoring [2]. These and other applications depend on very small, long-lived nodes that can be deeply embedded into the physical environment in ways that go beyond sensor nodes and approach the original vision of “smart dust” [28].

The research agenda associated with CRFIDs is now becoming defined as the community uses prototype tags to experiment with applications [3, 6, 9]. A fundamental problem for these devices is the efficient use of energy. Energy is the scarce resource that limits the amount of computation that can be performed because it must be harvested at low rates from signals transmitted by readers meters away. Further, to remain physically small and to power-up quickly, CRFIDs have miniscule energy stores compared to sensor network nodes. For example, the energy store of the WISP [24] prototype tag is *eight* orders of magnitude smaller than the battery of the popular Telos sensor mote[18]. This means that CRFIDs will typically exhaust and recharge their energy stores many times a second. In turn, it means that runtimes for sensor networks are of little use for CRFIDs. Sensor node runtimes seek to keep long-term expenditures below long-term harvesting or to maximize node lifetimes measured in days [14]. In contrast, CRFID runtimes must take a short-term view to match lifetimes measured in milliseconds.

The problem we tackle in this paper is how CRFID tags can make efficient use of the available energy. The naive RFID power model on which CRFIDs are based is for the tag to turn on and run whenever it is powered by the reader. This approach works for traditional RFID tags because tag functionality is very simple (a state machine with memory) and can be run in the worst case at the limit of the energy harvesting range. However, CRFID tasks consume greater energy with more complicated tasks that use sensors and computation. By adopting the model of running whenever there is power, current CRFID designs reduce the range at which a CRFID tag functions and limit the kinds of tasks that can be run. Prior work has looked at tuning the CRFID hardware constants (e.g., capacitor sizes) to better match available energy to a specific task [8]. Instead, our approach is to view the need to match harvested energy to task consumption as a scheduling problem. We wake the tag out of deep sleep only when it is likely to execute a task efficiently. This enables devices to run a range of tasks efficiently without requiring hardware modications.

We present the design and evaluation of *Dewdrop*, an energy-aware runtime for CRFID tags. We have implemented *Dewdrop* on the Intel WISP tag, and have experimented by powering the tags using a commodity Impinj UHF RFID reader for a range of distances, number of competing tags, and light and heavy CRFID tasks. By waking tags at the right times, we find that we can run tasks where they previously could not run, and about as often as possible given the energy that the RF environment provides. Prior to our work, the WISP had an oper-

ating range sufficient for point demonstrations. With our runtime, it is possible to use a single RFID reader to track CRFID tags on everyday objects in a room with enough responsiveness for activity inference.

While *Dewdrop* is conceptually simple, we found a practical design difficult to achieve for several reasons. First, the energy needed to run a task and the input RF power both vary greatly over time due to factors such as non-deterministic protocols and reader frequency hopping. This hampers predictions of when to start the next task execution. Second, our intuition about energy storage as a simple reservoir proved wrong because a fixed amount of energy is more or less expensive to store depending on when it is gathered, and the rate at which it is consumed depends on when it is spent. This leads us to track other forms of waste. Finally, it is costly to gather the basic information needed to make scheduling decisions because CRFIDs are so energy impoverished. This required opportunistic measurement strategies and careful implementation.

We make three contributions. First, we formulate the task scheduling problem for CRFID tags with limited energy storage. Second, we present the design of a runtime that enables CRFID tags to adapt their behavior to best match task energy requirements to available energy over the factors that most affect efficiency. Third, we show by experimentation with the WISP tag and an Impinj RFID reader that our design is much more effective than prior techniques for real energy costs and RF conditions. *Dewdrop* doubles the operating range for heavyweight tasks as compared to the WISP hardware that runs tasks whenever there is power, and keeps overhead low to match the performance for lightweight tasks to which the WISP hardware is well suited.

The rest of this paper is organized as follows. We start with background in Section 2 and then define the task scheduling problem for CRFIDs in Section 3. We present the design of *Dewdrop* and its implementation in Sections 4 and 5. Our experimental evaluation is in Section 6. We follow with related work in Section 7 and conclude in Section 8.

2 Background

We begin with relevant background on computational RFID because it is an emerging research area.

CRFID tags and the WISP. CRFID tags combine RFID technology for energy harvesting and backscatter communication with computation and sensing. The prototype CRFID tag that we use is the Intel Wireless Identification and Sensing Platform (WISP) [24]. Other prototype CRFID tags exist [21, 30], but the WISP is the most widely used because it is available to the academic community.[1]

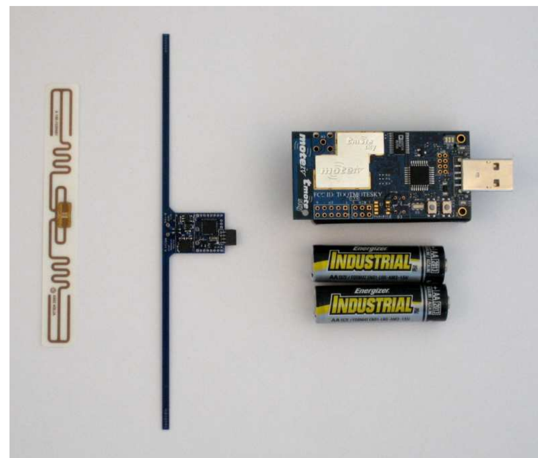


Figure 1: Gen 2 tag, Intel WISP, Telos mote.

Figure 1 shows the WISP in comparison to a Gen 2 UHF RFID tag and a Telos mote. Like an RFID tag, it is small, thin, and battery-free. It runs only when powered by energy harvested from an EPC Gen 2 RFID reader and communicates with the reader using a low-energy form of signaling called backscatter. The current WISP can harvest sufficient power to operate at up to 4m. As advances in processor and sensor technology continue to reduce power consumption, the range of WISP tags will increase accordingly.

Like a very low-end mote, the WISP is fully programmable, capable of running small programs, and equipped with sensors. The WISP runs programs written in C on an ultra-low power 16-bit MSP430 microcontroller and has 8K of flash memory, a 3D accelerometer, and temperature and light sensors.

However, unlike an RFID tag, the WISP consumes considerably more power when computing, communicating and sensing than can normally be harvested from the reader signal. Consequently, the WISP must duty cycle between a low-power sleep mode, in which the energy needed to run is gathered into a short-term energy buffer, and an active mode in which stored energy is consumed.

We expect future CRFID tags to be more capable than the WISP, but to remain very-low end devices, even compared to sensor nodes. As the power efficiency of the devices improves slowly over time, so too will the sensing and processing demands that are placed on them; thus, the disparity between harvestable power and operating power will remain.

CRFID Applications. CRFID tags and readers are enablers for ubiquitous computing applications that benefit

[1] See `wisp.wikispaces.com` for open-source WISP software and hardware designs. WISPs are in use at more than 30 universities.

from instrumentation on or as part of objects in the physical world. For example, the WISP has been used to prototype applications for gesture-based access control [6], cold chain monitoring [29], and activity recognition for eldercare [3].

We delve into the last scenario to give one example of a workload that *Dewdrop* is intended to support. The automatic recognition of the activities of elderly people can improve quality of life by helping elders remain in their own homes for longer with inexpensive care. It does this by tracking key indicators of well-being such as medication adherence, mobility and exercise, food and water intake, changes in routine, and safety [17]. The use of CRFIDs for activity recognition can deliver a solution that is inexpensive and non-intrusive. CRFIDs with accelerometers can be affixed to objects in an elder's home, and data gathered from the tags can be used to determine activity. This has advantages over existing solutions as it requires neither monitoring by cameras, which can invade privacy, nor on-body sensors, which can be inconvenient for elders. Additionally, this type of deployment would be difficult using motes because of their size and cost.

In earlier work, we prototyped such a system by tagging objects an elderly person normally interacts with—her medicine cabinet, tea kettle, teacup, toothbrush, etc.—with CRFIDs with onboard accelerometers [3]. RFID readers were placed out of sight in the ceiling. Each CRFID repeatedly sampled its accelerometer and transmitted its value to the readers. The readers detected tags that moved by looking for changes in those values, for instance, when a CRFID-tagged medicine bottle is picked up. Activities such as *preparing a meal* and *taking medicine* were then inferred from sequences of object use.

We built our earlier system using WISPs and found that the system worked, albeit with a smaller coverage region and lower response rates than we expected. This meant that we needed to deploy multiple readers per room, and even then some tags responded infrequently, which degraded activity inference. After some investigation, we determined that the WISPs were wasting much of the available energy. That discovery led to our work on *Dewdrop*.

3 Problem

Our goal is to run programs on CRFID tags in a way that makes the best use of the available energy, which in turn extends operational range and increases responsiveness. In this section, we formulate this goal as a scheduling problem and describe the key challenges.

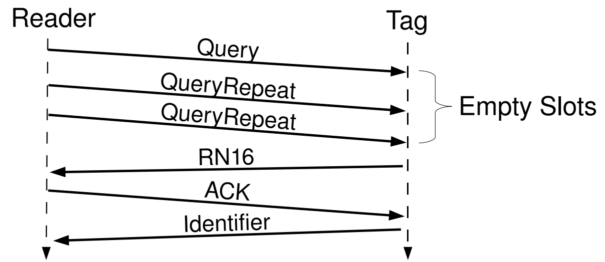


Figure 2: Example message exchange of a reader identifying a tag.

3.1 Task Model

In our setting, a reader powers one or more nearby tags and requests that they perform tasks. Tags may come and go from the range of a given reader as the RF environment changes or the tag or reader moves. In keeping with other CRFID and RFID applications, we assume that each CRFID tag repeatedly executes a single fixed operation as often as possible (e.g., reporting a sample), but from time to time may be retasked to perform a different operation (e.g., switch from sampling the accelerometer to measuring the light level). Additionally, tags in the deployment may be executing different tasks. As a tag considers only one type of task at a time, scheduling the order and execution of multiple tasks on a single tag is both unnecessary and out of scope.

We define a *task* to mean a short program that is run to completion without pause. While it may be possible to break some tasks into phases, the timing requirements of the tag hardware, the RFID protocol, and application requirements make it impractical to interrupt many tasks once they start. Due to the operating constraints of a tag, tasks are fairly inflexible and have limited functionality. They can support modest processing, e.g., for lightweight encryption, but generally consist of sensing and reporting operations. Even with this limited task diversity, tasks have very different power requirements. For example, measuring the light level consumes much less power than activating and sampling the accelerometer. We experiment with examples at the lower and higher ends of this spectrum later in the paper.

We assume that CRFID tags will be powered by a standard Gen 2 RFID reader, at least in the near future. This is likely, as it allows CRFID tags to take advantage of deployed and commodity infrastructure. Tasks often return a result to the reader. Contention between the transmissions of multiple tags is managed by the EPC Gen 2 MAC protocol [7] that is based on Framed Slotted Aloha [25]. To gather tag IDs, the reader transmits a *Query* command that indicates the number of slots in the frame. Tags then randomly choose a slot in which to reply, and transmit a 16-bit random number in their slot.

The reader *ACKs* this random number and the tag replies with a 96-bit identifier. An example of this exchange is shown in Figure 2, where no tag chooses the first two slots, and one tag responds in the third slot. Tags that collide in a slot are not *ACK*ed and respond again after the next *Query*. The reader iteratively modifies the frame size to best match the number of tags that are present. Sensor and other data is transferred on top of this protocol, either by overloading the identifier bits or using further commands that read and write tag memory. New MAC protocols specially designed for CRFIDs are also of interest, but we leave them to future work.

3.2 Task Scheduling Goal

Given that tags repetitively execute a task whenever possible and the reader power is not controlled by the tags, maximizing energy efficiency is equivalent to maximizing the rate at which tasks successfully complete. We use task completion rate, in terms of how many task iterations succeed over a given time period, as a metric to evaluate the performance of *Dewdrop* in the steady state. Since energy falls off with distance (at least as quickly as distance squared), we expect the completion rate to fall with distance. But, it should not fall more quickly than the available energy.

CRFID tags like the WISP collect the energy harvested from RF signals into a capacitor that matches the fluctuating input power to the steady output power needed to run the tag. Energy is harvested whenever a nearby reader is transmitting an RF signal. Like an RFID tag, the WISP hardware begins task execution whenever a fixed, hardware-defined power level that is sufficient to activate the tag is reached. Once a task iteration has started, it may either run to completion or fail if the CRFID tag runs out of energy first. We use this fixed, hardware approach as a baseline for comparison in our evaluation.

Dewdrop replaces the fixed, hardware approach with an adaptive software strategy. There is only one decision that a tag can make to improve energy efficiency: to defer the start of a task it could otherwise begin, sleeping until the energy store becomes more full. This is useful because the larger store of energy increases the chance that the task will run to completion. However, it is wasteful in terms of time and energy if the task would have succeeded anyway. The runtime's job is to decide when to run and when to sleep depending on the task and RF environment.

3.3 Challenge: Varying Task Needs

A good runtime will not start a task unless there is (likely) sufficient energy to complete it, as failing a task consumes energy without doing useful work. Yet whether a task will succeed is difficult to predict because task energy requirements vary greatly due to two main factors.

Different size tasks. The energy consumption of different tasks can vary widely depending on the sensors they use, the computation they perform, and their communication patterns. In our experiments, we consider a light task that simply takes an accelerometer reading, and a much heavier task that additionally uses the RFID communication protocol to send the accelerometer data to the reader by embedding it in the tag identifier. We refer to these as the SENSE and SENSETX tasks, respectively.

Non-deterministic tasks. Tasks may be non-deterministic, which causes their energy requirements to vary from execution to execution. An important source of non-determinism is the RFID MAC protocol. The number of messages that a tag must process to communicate with the reader depends on both the number of other tags present and the collisions that happen to occur. As a consequence of the way the protocol works, a tag that chooses to take part in a communication round must complete the transaction; it cannot sleep or it will lose synchronization with the reader. Other sources of non-determinism may come from sensor data itself, the timing of reader queries (which a tag cannot control or predict) or random numbers used in security protocols.

3.4 Challenge: Platform Inefficiencies

The variation in task energy requirements suggest that a better strategy might be to overestimate the task needs. For example, a tag could harvest energy until its buffer is completely full before executing a task. In this way, it would run with “a full tank” to avoid preventable failures and top off between tasks. Unfortunately, storing excess energy is wasteful due to platform characteristics.

Sublinear charging. CRFIDs use capacitors for energy storage as they are well suited to energy harvesting devices [12]. They charge quickly, recharge indefinitely, are small and inexpensive, and are non-toxic. However, capacitors store energy faster when they are close to empty than when nearly fully charged. This nonlinearity is fundamental to the way capacitors work. As the capacitor voltage, which increases with increasing charge, approaches the voltage supplied by the energy harvesting circuitry, the charging current decreases to zero. Thus, to increase the task rate, it makes sense to operate with a lightly charged capacitor.

Superlinear discharge. Regulating circuitry must adjust the supplied (input or stored) voltage to the operating voltage. Differences in voltage levels inevitably lead to some voltage-dependent conversion losses. For exam-

ple, the WISP uses a linear regulator that sheds the voltage difference by dissipating heat, which wastes energy. Other techniques are possible but come with their own tradeoffs (e.g., switching regulators are more efficient but have greater leakage, don't work when the input voltage is near the target voltage, and are inefficient when they start up[2]). To minimize energy wasted while discharging, the tag again should operate with its capacitor at a minimal charge.

The exact inefficiencies will vary with the CRFID, but we believe that all real platforms will have these kinds of nonlinearities. The implication is that a quantum of energy may cost (or be worth) a different amount depending on when it is gathered (or spent), with excess energy being more wasteful.

3.5 Challenge: Varying Input Power

Even assuming that the tag runtime could accurately estimate tasks costs, it is difficult to know how long to sleep to store sufficient energy because the rate at which a tag harvests energy changes over time.

Widely varying input powers. RF power received at a tag decreases at least as fast as the square of its distance from the reader. In practice, this means that the available energy varies by more than an order of magnitude over useful ranges. Hardwiring tags to operate at the low end of the power scale wastes a significant opportunity at the high end of the scale, and restricting tags to operate at the high end of the scale limits operational range. Additionally, CRFIDs harvest energy even when the task is being executed. When the tag is close to a reader less energy will be drained from the energy store than when further from the reader. Consequently, when close to the reader, less energy needs to be stored before execution can begin.

Frequency selective fading. RFID systems operate in the 900MHz ISM band, so the reader must frequency hop every 400ms to obey FCC regulations. Multipath effects result in different frequencies being attenuated differently. This means that the received power at tags can vary widely over short time scales.

4 Design

We now develop the design of our energy-aware runtime, *Dewdrop*. The main scheduling decision is when to start the next task iteration. Starting too soon wastes energy when the tag runs out of power and the task fails. Starting too late collects excess energy, which is inefficient to both store and use. Our approach is to minimize both forms of waste. As we develop our design, we present microbenchmarks using the WISP to show the importance of the different factors we identified as challenges.

[2]This and other parts and design tradeoffs make the linear regulator the best choice for the WISP.

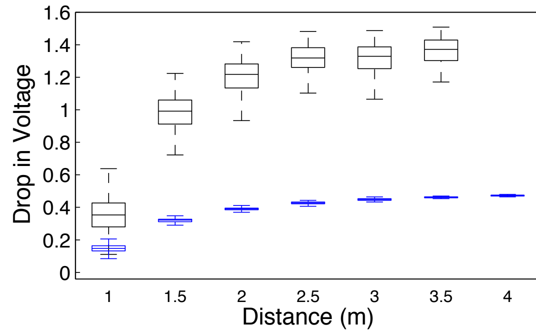


Figure 3: Voltage drop forSENSETX (upper black items) and SENSE (lower blue items).

4.1 Design Goals

From our problem formulation, the overarching goal of *Dewdrop* is to convert all available energy into completed task iterations. This goal is equivalent to two sub-goals that help to enable new applications:

Increased range. We want our runtime to execute a task at greater distances from the reader than the baseline WISP hardware. Each task should work from next to the reader out to the distance at which the tag can no longer harvest enough energy for the task.

Improved responsiveness. At all distances, we want to increase responsiveness compared to the baseline WISP hardware. We never want to noticeably decrease responsiveness.

Both goals are met by maximizing the task completion rate for a given task and distance from the reader. In practice, achieving them implies that we must meet two other goals:

Low overhead. The implementation of *Dewdrop* must be extremely lightweight. Operations such as checking the level of the energy store or calculating sleep periods consume scarce energy. Even a modest amount of overhead can easily negate the benefits of scheduling tasks.

Adaptation. Tags must operate well across a range of deployment scenarios. For example, they may be configured to run either heavy or lightweight tasks, and they must run their task efficiently both when near and far from a reader. Our performance sub-goals are stated across these factors, so *Dewdrop* must adapt to the environment at runtime.

4.2 Variation in Task Costs

To predict when to start a task, *Dewdrop* must estimate how much energy the task will need over and above the energy that will be harvested by the tag while it runs the task. This depends on the factors we previously identified: the task itself, other tags competing for the medium, the distance from the reader and the frequency on which the reader is transmitting, and the amount of energy already in the capacitor. All of these factors are fundamental. However, they may differ in magnitude with implications for system design. For example, if the energy needs depend mostly on the type of task, then each task could be profiled offline to characterize its fixed energy need.

To understand how much these factors matter in practice, we ran an experiment with the SENSE and SENSETX tasks running on a WISP. For the WISP, the energy consumption of a task can be measured by the drop in the voltage of the capacitor that acts as a short-term energy buffer[3]. Figure 3 shows this voltage drop as a function of distance for the two tasks. Box plots show the distributions over at least 300 task executions at each distance.

The SENSE task is deterministic. However, we see that the voltage drop is significantly larger when the tag is far from the reader than when it is close to the reader; it more than triples. This is because the input power from the reader varies by more than an order of magnitude. A second effect is that the variance is larger when the task is run close to the reader because the input power supplements stored energy and varies with the reader transmit frequency. At 1m this variance is approximately 0.3V compared to 0.1V at 4m.

Looking at the SENSETX task, the drop in voltage is almost three times larger than for SENSE. At 4m, the WISP cannot store sufficient energy to execute the task[4]. The variation is also higher at all distances because this task is non-deterministic. Its energy consumption depends on randomization in the Gen 2 MAC protocol, and the variation would be even greater if there were multiple WISPs (which we study as part of our evaluation).

These results imply that *Dewdrop* should adapt to both the task and the environment in which the tag is operating. Any fixed energy target at which to start a task will be either too low, causing the tag to fail at a distance when it could still run, or too high, causing the tag to run tasks more infrequently than it is capable of sustaining. A second implication is that it is likely not feasible to accurately estimate the energy needs of a particular task execution due to inherent variation. Instead, *Dewdrop* must adapt an estimate of energy needs that captures the effects of the distribution.

[3]The energy stored in a capacitor is calculated as $\frac{1}{2}CV^2$, where C is the capacitance and V is the measured voltage.

[4]To even run the task over a range of distances we needed to modify the baseline WISP behavior.

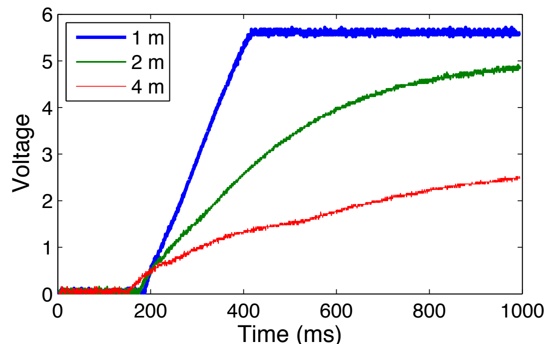


Figure 4: WISP capacitor voltage over time

4.3 Minimizing Wasted Energy

Sources of waste. Energy is wasted when the CRFID tag starts too early and fails to complete the task, or waits too long and inefficiently collects excess energy. How much energy is wasted in these cases depends on how CRFID tags convert reader energy into harvested energy and consume this energy.

To gain some insight, we performed a simple experiment by charging a WISP without running any task. Figure 4 shows the voltage of the WISP capacitor as it charges at different distances. (The RF source powers on at approximately 200 ms.) This is the expected behavior. A capacitor's charging rate decreases by a factor of e every RC seconds, where R and C are the resistance and capacitance of the RC circuit and e is the base of natural logarithms, and asymptotically approaches zero as the capacitor charges to the voltage of the power source.

This charging behavior has two implications. First, it shows the effects of distance. Far from the reader, the low received power limits the maximum energy that can be stored. At 4m the capacitor approaches only 2.75V, while at 1m it rises quickly to 5.8V (at which point an over-voltage protection circuit kicks in). This means that heavy tasks will not run as far from the reader as lightweight tasks no matter how long the tag sleeps.

The second implication is that, even for a fixed input power, it is inefficient to charge to a higher voltage than necessary. Because the rate at which energy accumulates in a capacitor decreases exponentially as it charges, storing excess energy wastes *time*. There is a penalty for charging too high and leaving spare energy in the capacitor. In a sense, that leftover energy was "cheaper" to store. This effect is magnified by the linear regulator of the WISP, which consumes more power when there is a higher charge on the capacitor.

To capture these factors, *Dewdrop* estimates waste in

terms of time. This directly accounts for the energy consumed by a task, even if it fails, and also for how long it took to store that energy. While the details will differ, all platforms are likely to have nonlinearities with respect to storing and consuming energy that make it useful to measure waste in terms of time. For instance, capacitors are the natural choice for short-term energy storage, and all CRFIDs that use capacitors will have this kind of inefficiency.

Balancing sources of waste. Intuitively, starting tasks later, at a higher energy level, will decrease the time wasted due to tasks failing but increase the time wasted due to excess charging. Our goal is to minimize the total wasted time due to both causes. Since the energy cost of executing a task cannot be estimated precisely, *Dewdrop* aims to reduce the expected wasted time in the following manner. Let $P(fail|V_s)$ be the probability that the task will fail given a starting voltage level V_s. The runtime's job is to choose a V_s in the range $[V_0, V_{max}]$ that minimizes the wasted time:

$$\begin{aligned} t_{wasted}(V_s) &= P(fail|V_s)t_{under} \\ &+ (1 - P(fail|V_s))t_{over} \end{aligned}$$

where t_{under} is the time to charge back to V_s after a failure and t_{over} is the time spent overcharging, i.e., the time spent charging beyond the energy level that would have been sufficient. Note that this implies that some rate of failures may be desirable as charging high enough to assure success incurs a penalty that accumulates on every execution.

A naive approach to finding the V_s that minimizes wasted time would be to try every value of V_s. This is impractical, as the tag would need to examine a sufficiently long series of task execution attempts at each V_s to determine which had the best performance. Furthermore, this search would need to be repeated periodically as the RF environment and other factors change.

To avoid this search, we use our intuition that the two kinds of wasted time tradeoff against each other to find an approximate solution. Let P_f be the current task failure rate at a fixed starting voltage V_s and $T_{under} = P_f * t_{under}$ and $T_{over} = (1-P_f)*t_{over}$. If $T_{over} >> T_{under}$, then the runtime is too conservative; it could have chosen a lower V_s. If $T_{under} >> T_{over}$ then it is being too aggressive; V_s is too low and tasks are failing too often.

Dewdrop uses the heuristic that balancing the two sources of waste tends to minimize overall wasted time; this at least finds a reasonable operating point by ensuring that neither factor is a major source of inefficiency. Additionally, tracking and comparing the two sources of wasted time requires minimal computation which is key for any viable solution. The balance point can be found by slowly updating V_s to trade T_{under} against T_{over}. To do this, *Dewdrop* maintains separate estimates of T_{under} and T_{over} that are updated with an exponentially weighted moving average (with parameter α) each time a task executes depending on its success or failure. The two estimates are then compared, and the energy level V_s is adjusted by β in the direction that will balance the averages. That is, it is increased if more time is being wasted on failures than on charging too high.

More precisely, let V_e be the voltage at the end of running a task, and V_0 be the voltage at which the tag ceases to operate, and ϵ be a small voltage. A task succeeds if and only if $V_e \geq V_0 + \epsilon$. *Dewdrop* computes estimates and uses them to adjust the target energy level, V_s as follows:

$$T_{over} = \begin{cases} (1-\alpha)T_{over} + \alpha t_{over}, & \text{if } V_e \geq V_0 + \epsilon \\ (1-\alpha)T_{over}, & \text{if } V_e < V_0 + \epsilon \end{cases}$$

$$T_{under} = \begin{cases} (1-\alpha)T_{under}, & \text{if } V_e \geq V_0 + \epsilon \\ (1-\alpha)T_{under} + \alpha t_{under}, & \text{if } V_e < V_0 + \epsilon \end{cases}$$

$$V_s = \begin{cases} V_s - \beta, & \text{if } T_{over} > T_{under} \\ V_s + \beta, & \text{if } T_{under} > T_{over} \end{cases}$$

Of course, there are degenerate cases where this heuristic will fail, e.g., tasks that exhibit bimodal energy consumption where some executions consume a lot of energy and some executions consume very little. But, based on applications we have seen in the literature, our approach is a good fit and has the benefit of being both simple and efficient.

4.4 Charging to a Target Energy Level

Given a target energy level, the CRFID runtime must arrange for the task to begin execution when stored energy reaches that target. The baseline WISP uses hardware support in the form of a voltage supervisor to start execution when the capacitor voltage reaches a fixed level of 2V. Unfortunately, there are no designs for variable voltage supervisors that can be used in CRFIDs to the best of our knowledge.

Instead, *Dewdrop* uses a software polling approach to determine when the target energy level has been reached and execution should begin. It sleeps while energy is being harvested, and occasionally wakes up to sample the capacitor voltage using an analog to digital converter (ADC). This is a general strategy that can be used on most platforms regardless of how the target energy level is determined.

However, polling is difficult to achieve at low cost because charge times can vary over orders of magnitude

and waking up and sampling the capacitor consumes precious energy. In our experiments with the WISP, we found that reaching a given threshold can take less than 10ms or 100s of ms depending on the input power. This variation, combined with the non-trival cost of waking up to take a sample, means that polling at any fixed interval is problematic. If the tag is close to the reader, a long interval means that the tag will store excess energy and miss opportunities to execute tasks. Conversely, if the tag is far from the reader, it will accumulate energy very gradually and pay a disproportionately greater overhead if the interval is short.

To gather energy over a large range of input powers and target voltages, *Dewdrop* uses an exponentially adapted polling interval. Specifically, let V_r be the voltage a tag has gained since it last woke up, and t be the current sleep interval. Then,

$$t_{next} = \begin{cases} 2t, & \text{if } V_s - V > 2V_r \\ t/2, & \text{if } V_s - V < V_r/2 \\ t, & \text{otherwise.} \end{cases}$$

This mechanism is very lightweight because it only involves shift operations to scale the polling interval, not multiply, divide, or floating point operations (which are not likely to be available in hardware). In our evaluation we find it to be responsive, sleeping for short amounts of time at high input power, and to have low overhead, gathering energy out to low input power levels.

5 Implementation

The WISP firmware is written in a mix of C and assembly, for timing sensitive operations. The code can be broken down into two main components: the *Dewdrop* runtime and task support. The *Dewdrop* runtime code must execute quickly and infrequently to reduce overhead. Task support includes the Gen 2 RFID communication protocol, which requires tags to respond to reader commands quickly, generally within 10s of microseconds. This section describes our implementation of a functioning prototype as it relates to these challenges.

5.1 WISP Hardware

The WISP draws approximately 600μA when the CPU is in active mode and 1.5μA when in a state-preserving sleep mode. By default, the WISP wakes up at a fixed power level; a voltage supervisor waits for sufficient power to operate (defined by its capacitor reaching 2V) and then triggers a hardware interrupt to wake the device. We use the term *HwFixed* to refer to this hardware method of waking up at a fixed voltage. *Dewdrop* disables this mechanism and instead uses a timer interrupt to wake the device.

The WISP stores energy in a 10μF capacitor and the voltage of the capacitor can be sampled via its analog to digital converter.[5] If the voltage of the capacitor drops below 1.5V, the WISP will black out and lose all state. We found that the time to fully charge the capacitor varied from 10s to 100s of milliseconds, depending on distance. Discharging a full capacitor to below 1.5V in the absence of a reader signal takes 10s of ms when active, but more than 8s when in sleep mode. Thus, the WISP can carry state across relatively long periods of reader inactivity by sleeping.

5.2 *Dewdrop*

Low power wake-up. *Dewdrop* puts the WISP into a deep sleep state for a specified period to gather energy, and the CPU is woken up by the timer interrupt. The process is repeated until the target wake-up voltage, V_s, is reached. This approximates the behavior of a hardware voltage supervisor, which wakes a device when a specified voltage is reached, but allows us to vary V_s. A potential drawback to this approach is an increased current draw due to keeping the crystal oscillator active to drive the timer, but in practice this increase is acceptably small (2 μA vs 1.5 μA with the crystal off).

Low cost voltage sampling. *Dewdrop* checks the capacitor voltage to see if enough energy has been stored to warrant starting a task, and goes back to sleep if not. The energy overhead of this polling approach is determined by the polling interval and how long the WISP must be awake for each sample. The per sample cost is directly proportional to how long the WISP must stay in active mode. Sampling the capacitor voltage should take 90μs according to the MSP430 data sheet instructions for using the ADC. However, we found that ADC values stabilized much faster—20μs including setup time—with sufficient accuracy (10mV). This shorter awake time drastically reduced the cost of voltage sampling.

Calculating the energy storage rate. *Dewdrop* also tracks how quickly energy is being stored, as it uses this information to adapt the sleep period and to calculate how much time is wasted overcharging. Our adaptive sleep function generally results in a series of sleep periods, where the WISP wakes up and checks its voltage, adjusts the sleep period, and returns to sleep. When a task completes, $V_e - V_o$ tells us how much energy is leftover. We use the last period's charging rate and the average charging rate over all periods to estimate how much time was wasted overcharging. When a task fails,

[5] A 10 μF capacitor is a reasonable trade-off between charge time (a smaller capacitor charges faster) and charge capacity.

$V_s - V_o$ tells us how much energy was wasted. We use the average charging rate to calculate the time wasted undercharging.

5.3 Task Support

Order of operations. The computation and sensing components of tasks must take place before or after communicating with the reader; the deadlines imposed by the Gen 2 protocol are too tight to interleave task processing and message handling. Therefore, in the SENSETX task, for example, the WISP samples the sensor immediately after waking up and then begins decoding reader commands and waiting for the next Query.

Detecting task failures. To avoid blacking out and losing state, the WISP needs to detect when task failures are imminent and then quickly enter sleep mode. In other words, if the voltage drops below $V_o + \epsilon$ (see Section 4), the task must be aborted. In future hardware revisions of the WISP, we would like to trigger an interrupt when a *minimum* voltage threshold is reached. In the meantime, we approximate this behavior by manually inserting calls to the voltage sampling function in the task code. We found that an ϵ of 0.15V was sufficient to protect against blackout. That is, if any voltage sample measures below 1.65V, the WISP will sleep and record a task failure.

Sampling the voltage during the communication phase proved difficult, but it was necessary because message processing is a major factor in energy consumption. The Gen 2 message timing constraints are such that the WISP does not have time to take a sample between messages without losing synchronization with the reader, even with a sampling time of only 20μs. However, we found that we could carefully schedule a voltage sample during the preamble of every reader command, so long as the inspection of the sample was deferred until after the command was decoded. As the WISP must be in active mode to accurately track the preamble, this approach amortizes the cost of keeping the CPU active for decoding. This strategy makes it possible for us to closely track the voltage of the capacitor at every reader command with essentially zero overhead.

Randomness. The Gen 2 MAC protocol requires that tags choose slots randomly. As a source of randomness, we sample the voltage in the capacitor once immediately when the WISP first powers up, and use this value as a seed for a pseudo-random number generator. The variance in this voltage sample, due to input power and noise in the ADC, gives us sufficient randomness. Alternatively, we could have used SRAM state as a random source, with similar efficiency [11].

5.4 Monitoring Support

Monitoring WISP state and operation for debugging and experimentation is difficult. Traditional methods for debugging embedded systems, such as a JTAG connection, would supply power to the WISP and change its behavior. Instead, we use a custom monitoring board we developed for debugging WISPs [19]. The board communicates with a PC via USB, attaches to the debug and other output pins of the WISP, but does not add to or consume energy harvested by the WISP. The monitor board can also sample the voltage in the WISP's capacitor. For our study, we instrument the WISP to toggle debug pins at key points in its operation, and the monitor board records what event happened and immediately samples the WISP capacitor to determine its voltage. This results in a trace of WISP operations from which we can determine task costs, and response rates even for tasks that do not communicate with the reader.

6 Evaluation

In this section, we evaluate *Dewdrop* experimentally. We show that our approach of balancing sources of waste generally achieves 90% of the best possible response rate for the SENSETX and SENSE tasks and across a wide range of RF environments. *Dewdrop* improves performance over the default WISP runtime, providing applications a benefit in terms of both improved coverage and higher response rates.

6.1 Experimental Setup

Our experiments were conducted using an Impinj Speedway RFID reader that continuously transmits energy and commands. This is the normal reader behavior. For experiments involving a single tag, the WISP was placed on a poster board 1m from the reader antenna and the output power was variably attenuated from 30dBm (1 Watt), the maximum allowed for "Gen 2" readers, to 18dBm. This method increases repeatability by limiting the multipath effects that would occur if we moved the WISPs. We present results in terms of an equivalent distance that is calculated using free-space propagation, as we find them to be more intuitive than results in terms of transmit power.

In all experiments, we ran *Dewdrop* and the default WISP hardware, which we call *HwFixed*, that starts tasks at a fixed energy level of 2.0V. *HwFixed* provides a baseline for comparison. When possible, we also report results for *Oracle* as the best result found from an exhaustive offline search of starting energy levels (at which the WISP wakes-up and starts a task) using 0.03V steps. We

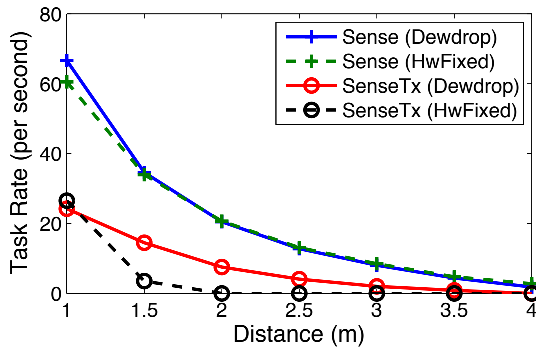


Figure 5: Response rates when using *Dewdrop* and the *HwFixed* runtimes.

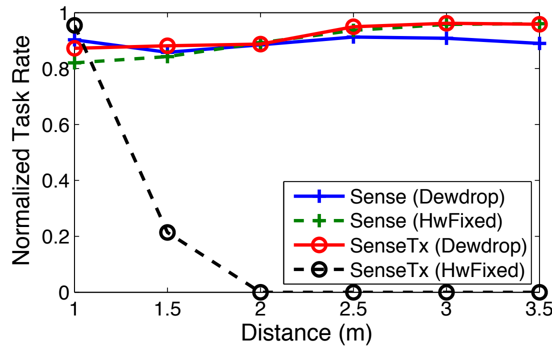


Figure 6: Response rates for *Dewdrop* and *HwFixed* compared to an oracle.

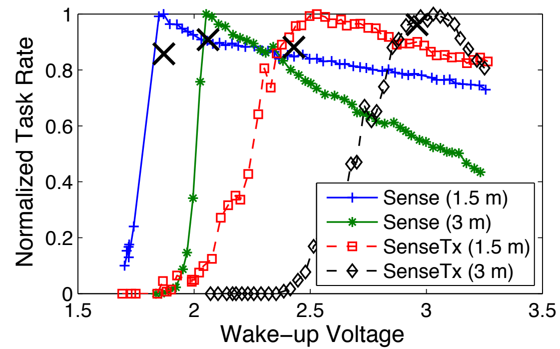


Figure 7: Response rates for both tasks at 1.5 and 3m. *X*'s indicate the operating point found by for *Dewdrop*.

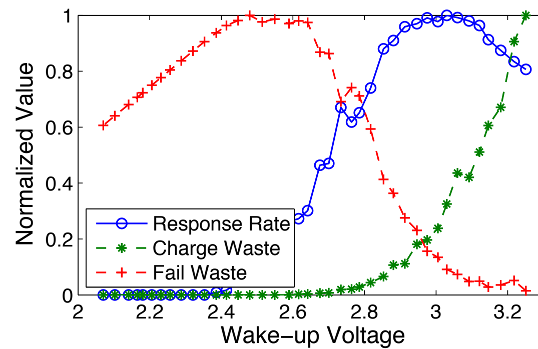


Figure 8: Response rate and wasted time for SENSE and SENSETX at 3m.

report results for both the SENSE and SENSETX tasks described in Section 4.2.

To evaluate our approach in a realistic deployment, complete with multipath effects, we deployed 11 WISPs with accelerometers on a 1.2m x .75m table of a model apartment at Intel Labs Seattle. This deployment is similar to that seen in [3], though we only consider a single workspace instead of the complete apartment. An RFID reader was installed in the ceiling and equipped with one antenna approximately 2m above the table pointing downwards. We configured the reader to run the SENSETX task to gather samples continuously for one minute. We performed three separate trials for each configuration to allow for variability from both the RF environment and communication protocol.

6.2 Using Energy More Effectively

***Dewdrop* performance.** We first assess how well *Dewdrop* performs compared to *HwFixed* for a single WISP.

Figure 5 compares the response rate of SENSE and SENSETX when using the two runtimes. *We find that the performance of* Dewdrop *consistently matches or exceeds that of* HwFixed. For the light SENSE task, the performance of *Dewdrop* closely matches that of *HwFixed* and actually performs better at 1m. This is because, at close range, the received power supplements stored energy enough to allow an energy level 0.2V below *HwFixed*'s fixed value.

In the case of the heavier SENSETX task, *Dewdrop*'s response rate decreases smoothly as reader power falls to 3.5m. *HwFixed* fails to execute the task beyond 1.5m. *Dewdrop* adapts to the higher energy requirements of this task, and stores more energy before beginning execution, whereas *HwFixed* does not. This improvement more than doubles the operating range of the tag.

To find an upper bound on how well *Dewdrop* could work, we compare to the *Oracle* results. Gathering this test data takes hours and is thus not a candidate for a practical CRFID runtime. Figure 6 again shows the response rates for the two tasks when using *HwFixed* and *Dewdrop*, but the rates are normalized by the best rates found using the *Oracle*. *We find that* Dewdrop *generally achieves better than 90% of the maximum rate seen by* Oracle *for both tasks*. Interestingly, *Oracle* always beat *HwFixed*. This means that the fixed 2 V energy level was never the best choice.

Evaluating *Dewdrop*'s choices. To understand why *Dewdrop* performs well, we looked at the starting energy

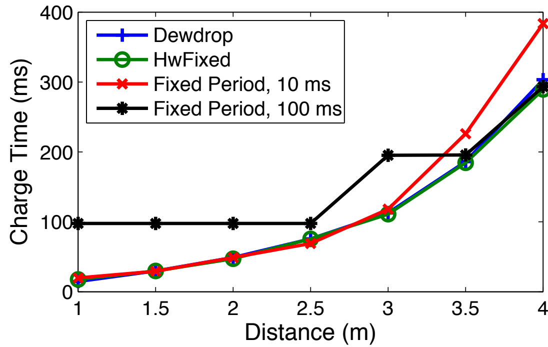


Figure 9: Charging time from 1.5V to 2V.

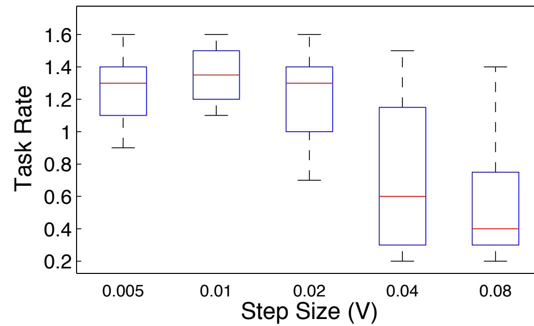


Figure 10: Effect of step size (β) on response rate for SENSETX at 3.5m.

levels it selects. *Dewdrop* must choose starting energy levels that are close to the best level found by the *Oracle* if it is to be efficient. To show that this is a non-trivial task, Figure 7 shows examples of response rate versus energy level curves. The figure is based on data from the *Oracle* for both tasks at 1.5 and 3m.

We see that the best starting energy level varies widely for different tasks and at different distances. For SENSE, the best energy level is 1.9V at 1.5m, when input power close to the reader supplements stored power, and 2.1V at 3m. Similarly, for SENSETX the best level varies from 2.5 to 3V over the same distance. These results emphasize that no fixed threshold will work either for all tasks or for all distances. For example, the best energy level for SENSETX at 3m is 3V. This level achieves only 50% of the maximum response rate for SENSE at the same distance. It is even worse if the best level for SENSE at 3m is chosen, as SENSETX cannot execute the task even once at 3m with an energy level of 2.1V.

The figure also shows the operating points found by *Dewdrop* marked with *X*s. *We see that our runtime finds points very close to the best energy level despite the differences between response curves*. Across all of our data the energy levels found by *Dewdrop* were within 0.1V of the best level found by *Oracle*.

To see how *Dewdrop* selects a good starting energy level, we looked at how it minimizes wasted time. We calculated the average wasted time per task due to failing and due to charging too high. Figure 8 shows this data, along with response rate, for an illustrative case of SENSE and SENSETX at 3m. The data are normalized by their maximum values. We see that as the starting energy level increases, the average wasted time due to failing generally decreases. (The waste is low at low wake-up thresholds despite tasks failing a greater fraction of attempts. This is because waste is computed in terms of time spent charging, and at low wake-up thresholds, very little time is spent charging.) Beyond 2.6V, waste from failed tasks decreases, as the task fails less often. Conversely, the wasted time from overcharging increases with the starting energy level because the energy is stored less efficiently at higher voltages.

Dewdrop seeks the intersection of the two waste curves, and uses the corresponding energy level. This appears to be a good strategy as the maximum response rate in the figure occurs near the intersection. Moreover, since the rates plateau around the maximum, *Dewdrop* can miss its mark by a fairly wide margin ($\pm$0.1V), without affecting performance significantly. Though the figure shows only a single example, *we found the energy level that equalized the two sources of waste generally achieved better than 95% of the maximum rate for both tasks at all distances*.

Evaluating *Dewdrop*'s costs.

This section investigates two possible inefficiencies in *Dewdrop*: the cost of our timer-based adaptive sleep scheme, and the effect of our choice of step size for maintaining the starting energy level. We show that both are efficient, which is in keeping with our runtime performing almost as well as the *Oracle*.

To be effective, our runtime must not appreciably increase charging time. Figure 9 shows the median charging time from 1.5V to 2V for *Dewdrop*'s adaptive sleep mechanism, the hardware wake-up of *HwFixed*, and two strawman versions of our software controlled sleep mechanism that use fixed sleep periods.

We find that, at all distances, our adaptive scheme achieves a charge time within 5% of the charge time of the hardware mechanism. Moreover, as expected, its performance is good over a wider range of distances than schemes that do not adapt their sleep periods. For example, the fixed period of 100ms does well at 4m (1.3% longer than *HwFixed*), but performs poorly at close range (600% longer than *HwFixed* at 1m). Likewise, fixing the period at 10ms works well at close range, but incurs significant overhead farther away (32% at 4m).

The second potential source of inefficiency in our system comes from our choice of step size (β) when seeking the best starting energy level. In *Dewdrop*, upward pres-

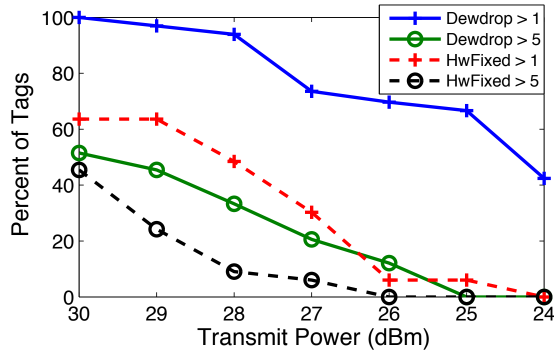


Figure 11: Percent of tags that have an average response rate above 1/s and 5/s using the two runtimes.

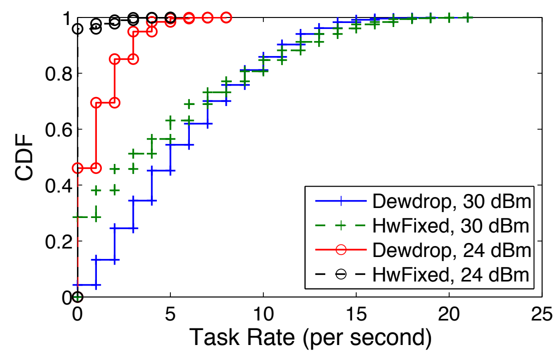


Figure 12: CDF of response rates for the two runtimes as power is reduced.

sure on the level is only exerted after it drops fairly low and tasks begin to fail; after failures, the starting energy level rises until the cost of overcharging outweighs the cost of failing. A small β increases the time it takes to adapt to environmental changes, while a larger β can result in large oscillations around the ideal wake-up threshold.

Figure 10 shows the effect of different step sizes on task rate for SENSETX at 3.5m. The average task rate per second is calculated over a 10 second sliding window. As step size increases, the task rates generally decrease and vary more widely. A larger step size means that *Dewdrop* increases/decreases its starting energy level too quickly, resulting in significant over/undercharging. The reverse then happens and the voltage is reduced by too much and more tasks fail. We found that a step size of 0.01V gave a good balance between damping oscillations in energy level and quickly adapting to environmental changes.

6.3 Multiple Tag Evaluation

Next, we evaluate *Dewdrop* in a realistic deployment consisting of multiple tags. To support CRFID applications such as activity recognition, our runtime should both increase the coverage region of the reader (e.g., so that distant devices respond) and also increase the response rates of the devices (e.g., so that object motion can more accurately be tracked). We consider both of these metrics for the 11 WISPs deployed in the model apartment.

Coverage. The coverage goal is to have as many devices as possible responding at a useful rate. Based on prior experience, we define two useful rates: a rate of 1/s, as is useful for low-rate object use detection; and a rate of 5/s, as is useful for higher-rate gestural recognition. To characterize the coverage of the deployment, the transmit power of the reader is reduced gradually to determine the "headroom" (in dBm) tags have for a given level of performance.[6]

We find that Dewdrop *has much better coverage than* HwFixed *because it enables tags to operate when much less incoming power is available.* Figure 11 shows the percentage of tags with average response rates above 1/s and 5/s when using the two runtimes. At 30dBm, all tags with *Dewdrop* respond at least once per second as compared to 64% with *HwFixed*. Coverage is better even when tags with *Dewdrop* receive one third the power of tags with *HwFixed* (viz., 67% for *Dewdrop* at 25dBm vs 64% for *HwFixed* at 30dBm). Moreover, at a four-fold reduction in power (24dBm), 42% respond with *Dewdrop* while none respond with *HwFixed*.

For a response rate of more than 5/s, the two runtimes perform equally well at 30dBm. This is because *HwFixed* works well when a tag receives good power from the reader. However, *HwFixed*'s coverage decays much more quickly with power than does *Dewdrop*'s coverage, e.g., at 27dBm *Dewdrop* has three times the coverage of *HwFixed*.

Response Rates. Figure 12 shows the distribution of the response rates of the tags when the reader is transmitting at 30 and 24dBm. The rates are computed over one second windows for both runtimes. *We find that* Dewdrop *consistently achieves higher rates, especially for the tags receiving less energy;* 30% of the data points are zero for *HwFixed* versus 5% for *Dewdrop*. *Dewdrop*'s ability to achieve useful rates is even more apparent when the reader transmits at 24dBm and tags are receiving one fourth as much power. *Dewdrop* obtains response rates greater than once per second 30% of the time, as compared to 2% with *HwFixed*. At 30dBm, *Dewdrop* and *HwFixed* achieve nearly the same rates for those tags that receive the most energy; 25% of the data points are above 9/s, and median rates are 5/s and 3/s respectively.

[6] This "attenuation thresholding" technique [10], has been shown to be more appropriate for characterizing RFID deployments than varying distance due to the high sensitivity of RFID to multipath.

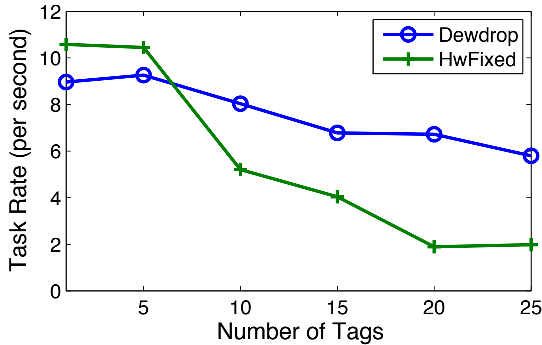


Figure 13: Response rate for the two runtimes as tag population size increases.

When more tags are present, the energy cost of communicating with the reader increases. This is because the reader increases the number of slots it uses to limit the likelihood of tag collisions, so CRFID tags must process more messages before transmitting to the reader.

Figure 13 gives the performance for a single tag when the reader transmits at 30dBm as additional tags are added to the deployment. The performance of *HwFixed* rapidly decreases with the number of tags. This is because the number of slots is increasing, and a tag cannot remain powered when it chooses a later slot. In contrast, *Dewdrop* simply increases its starting energy level to accommodate the additional communication overhead. With one tag, it wakes up around 2.5V whereas with 25 tags it wakes up closer to 3V. The result is that *Dewdrop* provides nearly three times the response rate as *HwFixed* when 25 tags are present.

7 Related Work

There has been significant work on building energy harvesting systems for sensor networks [27, 12, 1]. This work considers solar cells, but some conclusions apply equally to CRFIDs, e.g., [12] finds that capacitors should be used as the primary buffer to tolerate rapid charge/discharge cycles. In [26, 13, 15], the scheduling problem for energy harvesting devices is considered. The scheduling problem for these systems differs significantly from CRIFDs as they manage tasks and harvested power on the order of days, attempt to extend lifetime to months, and have no penalty for storing excess energy. In contrast, Dewdrop must store sufficient energy for a single task execution, and tolerate input power variations on the order of milliseconds in a context where every operation consumes precious energy.

Power management for CRFIDs has generally fallen into two categories; supplying additional energy and maintaining state information across power losses. Alternative methods of powering devices have been explored [16], with [5, 23] proposing solar cells and TV transmitters for CRFIDs. These approaches provide 10's of μW of supplemental power, an order of magnitude below the requirements of current CRIFDs, so energy still must be used efficiently.

In [20], the authors use offline profiling to estimate when state should be saved on the WISP, or transmitted to the reader [22], due to impending depletion of the energy store. We found that simply entering low power sleep mode is an effective way to maintain state, and it avoids the cost of writing to flash or transmitting to the reader in scenarios where the reader does not power off for long periods of time. In [8] the authors use offline modeling to help determine the appropriate capacitor size for a device designed to execute a particular task. While hardware modifications are necessary for tasks with dramatically different energy requirements, *Dewdrop* enables a wider range of tasks to be executed efficiently for any given energy store.

The WISP has been used to demonstrate power intensive applications that would benefit from our approach. RC5 cryptographic primitives were implemented in [4], and both cryptography and sensors have been used to increase the security of implantable medical devices [9], and credit cards [6]. For these applications, the energy requirements were far beyond what could be provided at range, and the studies were done using the WISP at close range. Dewdrop aims to enable such applications to operate more effectively at greater range.

8 Conclusion

We presented a runtime for CRFID tags that makes efficient use of the scarce available energy. Our runtime, *Dewdrop*, adapts a tag's duty cycle to match the harvested power to the sensing and computation cost of tasks. To do this, it estimates the time wasted by overcharging and by underestimating task needs, and uses the result to choose how much energy to buffer before starting a task. Using an implementation built on the WISP tag and a commodity RFID reader, we showed that *Dewdrop* runs tasks where prior techniques could not, and runs them at better than 90% of the best rate found by offline testing across a range of input powers, competing tags, and light and heavy tasks. *Dewdrop*'s adaptation effectively doubled the distance at which a tag executes tasks, which enables practical deployments. In an instrumented living space, all tags responded at useful rate to a single reader in the ceiling as compared to only 64% with fixed buffering. At over twice the distance (one quarter the transmission power), 42% of the tags still responded with *Dewdrop* while none responded with fixed buffering. We believe these performance levels bring us close to realizing a wide range of realistic CRFID applications.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd, Jason Flinn, for their helpful feedback. We would also like to acknowledge the invaluable assistance of Josh Smith and Alanson Sample in helping us understand the design and operation of the WISP. This work was supported in part by NSF award #1016487.

References

[1] D. Brunelli, L. Benini, C. Moser, and L. Thiele. An efficient solar energy harvester for wireless sensor nodes. In *DATE*, 2008.

[2] M. Buettner et al. Revisiting smart dust with RFID sensor networks. In *HotNets*, 2008.

[3] M. Buettner et al. Recognizing daily activities with RFID-based sensors. In *Ubicomp*, 2009.

[4] H. J. Chae et al. Maximalist cryptography and computation on the WISP UHF RFID tag. In *RFID Security*, 2007.

[5] S. S. Clark et al. Towards autonomously-powered CRFIDs. In *HotPower*, 2009.

[6] A. Czeskis et al. RFIDs and secret handshakes: Defending against ghost-and-leech attacks and unauthorized reads with context-aware communications. In *CCS*, 2008.

[7] EPCglobal. EPC radio-frequency identity protocols class-1 generation-2 UHF RFID protocol for communications at 860 mhz-960 mhz version 1.0.9. 2005.

[8] J. Gummeson, S. S. Clark, K. Fu, and D. Ganesan. On the limits of effective micro-energy harvesting on mobile CRFID sensors. In *MobiSys*, 2010.

[9] D. Halperin et al. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *IEEE Symposium on Security and Privacy*, 2008.

[10] S. Hodges et al. Assessing and optimizing the range of UHF RFID to enable real-world pervasive computing applications. In *Pervasive Computing*. Springer-Verlag, 2007.

[11] D. E. Holcomb et al. Initial SRAM state as a fingerprint and source of true random numbers for RFID tags. In *RFID Security*, 2007.

[12] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *IPSN*, 2005.

[13] A. Kansal et al. Power management in energy harvesting sensor networks. In *ACM Transactions on Embedded Computing Systems*, 2006.

[14] A. Mainwaring et al. Wireless sensor networks for habitat monitoring. In *WSNA*, 2002.

[15] C. Moser, D. Brunelli, L. Thiele, and L. Benini. Real-time scheduling for energy harvesting sensor nodes. *Real-Time Systems.*, 2007.

[16] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 2005.

[17] M. Philipose et al. Inferring activities from interactions with objects. *IEEE Pervasive Computing*, 2004.

[18] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-Low Power Wireless Research. In *IPSN/SPOTS*, 2005.

[19] R. Prasad, M. Buettner, B. Greenstein, and D. Wetherall. Wisp monitoring and debugging. In *Wirelessly Powered Sensor Networks and Computational RFID (to appear)*. Springer, 2011.

[20] B. Ransford et al. Getting things done on computational RFIDs with energy-aware checkpointing and voltage-aware scheduling. In *HotPower*, 2008.

[21] M. Reynolds and S. Thomas. The blue devil wisp: Expanding the frontiers of the passive RFID physical layer. *WISP Summit Workshop*, 2009.

[22] M. Salajegheh et al. CCCP: Secure remote storage for computational RFIDs. In *USENIX Security*, 2009.

[23] A. Sample et al. Experimental results with two wireless power transfer systems. In *IEEE Radio and Wireless Symposium*, 2009.

[24] A. P. Sample et al. Design of an rfid-based battery-free programmable sensing platform. In *IEEE Transactions on Instrumentation and Measurement*, 2008.

[25] F. Schoute. Dynamic frame length aloha. *IEEE Transaction on Communications*, 1983.

[26] J. Sorber et al. Eon: A Language and Runtime System for Perpetual Systems. In *SENSYS*, 2007.

[27] J. Taneja, J. Jeong, and D. Culler. Design, modeling, and capacity planning for micro-solar power sensor networks. In *IPSN*, 2008.

[28] B. Warneke, M. Last, B. Liebowitz, and K. S. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 2001.

[29] D. Yeager, P. Powledge, R. Prasad, D. Wetherall, and J. Smith. Wirelessly-charged UHF tags for sensor data collection. In *IEEE RFID*, 2008.

[30] D. Yeager, F. Zhang, A. Zarrasvand, and B. Otis. A 9.2a gen 2 compatible UHF RFID sensing tag with -12dbm sensitivity and 1.25vrms input-referred noise floor. *ISSCC*, 2010.

SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy

Anirudh Badam and Vivek S. Pai
Princeton University

Abstract

We introduce SSDAlloc, a hybrid main memory management system that allows developers to treat solid-state disk (SSD) as an extension of the RAM in a system. SSDAlloc moves the SSD upward in the memory hierarchy, usable as a larger, slower form of RAM instead of just a cache for the hard drive. Using SSDAlloc, applications can nearly transparently extend their memory footprints to hundreds of gigabytes and beyond without restructuring, well beyond the RAM capacities of most servers. Additionally, SSDAlloc can extract 90% of the SSD's raw performance while increasing the lifetime of the SSD by up to 32 times. Other approaches either require intrusive application changes or deliver only 6–30% of the SSD's raw performance.

1 Introduction

An increasing number of networked systems today rely on in-memory (DRAM) indexes, hashtables, caches and key-value storage systems for scaling the performance and reducing the pressure on their secondary storage devices. Unfortunately, the cost of DRAM increases dramatically beyond 64GB per server, jumping from a few thousand dollars to tens of thousands of dollars fairly quickly; power requirements scale similarly, restricting applications with large workloads from obtaining high in-memory hit-rates that are vital for high-performance.

Flash memory can be leveraged (by **augmenting** DRAM with flash backed memory) to scale the performance of such applications. Flash memory has a larger capacity, lower cost and lower power requirement when compared to DRAM and a great random read performance, which makes it well suited for building such applications. Solid State Disks (SSD) in the form of NAND flash have become increasingly popular due to pricing. 256GB SSDs are currently around $700, and multiple SSDs can be placed in one server. As a result, high-end systems could easily augment their 64–128GB RAM with 1–2TB of SSD.

Flash is currently being used as program memory via two methods – by using flash as an operating system (OS) swap layer or by building a custom object store on top of flash. Swap layer, which works at a page granularity, reduces the performance and also undermines the lifetime of flash for applications with many random accesses (typical of the applications mentioned). For every application object that is read/written (however small) an entire page of flash is read/dirtied leading to an unnecessary increase in the read bandwidth and the number of flash writes (which reduce the lifetime of flash memory). Applications are often modified to obtain high performance and good lifetime from flash memory by addressing these issues. Such modifications not only need a deep application knowledge but also require an expertise with flash memory, hindering a wide-scale adoption of flash. It is, therefore, necessary to expose flash via a swap like interface (via virtual memory) while being able to provide performance comparable to that of applications redesigned to be flash-aware.

In this paper, we present SSDAlloc, a **hybrid DRAM/flash memory manager** and a **runtime library** that allows applications to fully utilize the potential of flash (large capacity, low cost, fast random reads and non-volatility) in a transparent manner. SSDAlloc exposes flash memory via the familiar page-based virtual memory manager interface, but internally, it works at an object granularity for obtaining high performance and for maximizing the lifetime of flash memory. SSDAlloc's memory manager is compatible with the standard C programming paradigms and it works entirely via the virtual memory system. Unlike object databases, applications do not have to declare their intention to use data, nor do they have to perform indirections through custom handles. All data maintains its virtual memory address for its lifetime and can be accessed using standard pointers. Pointer swizzling or other fix-ups are not required.

SSDAlloc's memory allocator looks and feels much like the `malloc` memory manager. When `malloc` is directly replaced with SSDAlloc's memory manager, flash is used as a fully log-structured page store. However, when SSDAlloc is provided with the additional information of the size of the application object being allocated, flash is managed as a log-structured object store. It utilizes the object size information to provide the applications with benefits that are otherwise unavailable via existing transparent programming techniques.

Using SSDAlloc, we have modified four systems built originally using `malloc`: memcached [4] (a key-value store), a Boost [1] based B+Tree index, a packet cache

Application	Original LOC	Edited LOC	Throughput Gain vs	
			SSD Swap Unmodified	SSD Swap Write Log
Memcached	11,193	21	5.5 - 17.4x	1.4 - 3.5x
B+Tree Index	477	15	4.3 - 12.7x	1.4 - 3.2x
Packet Cache	1,540	9	4.8 - 10.1x	1.3 - 2.3x
HashCache	20,096	36	5.3 - 17.1x	1.3 - 3.3x

Table 1: SSDAlloc requires changing only the memory allocation code, typically only tens of lines of code (LOC). Depending on the SSD used, throughput gains can be as high as 17 times greater than using the SSD as swap. Even if the swap is optimized for SSD usage, gains can be as high as 3.5x.

backend (for accelerating network links using packet level caching), and the HashCache [9] cache index. As shown in Table 1, all four systems show great benefits when using SSDAlloc with object size information –

- **4.1–17.4** times faster than when using the SSD as a swap space.
- **1.2–3.5** times faster than when using the SSD as a log-structured swap space.
- Only **9–36** lines of code are modified (`malloc` replaced by SSDAlloc).
- Up to **31.2** times less data written to the SSD for the same workload (SSDAlloc works at an object granularity).

The rest of this paper is organized as follows: We describe related work and the motivation in Section 2. The design is described in Section 3, and we discuss our implementation in Section 4. Section 5 provides the evaluation results, and we conclude in Section 6.

2 Motivation and Related Work

While alternative memory technologies have been championed for more than a decade [10, 25], their attractiveness has increased recently as the gap between the processor speed and the disk widened, and as their costs dropped. Our goal in this paper is to provide a transparent interface to using flash memory (unlike the application redesign strategy) while acting in a flash-aware manner to obtain better performance and lifetime from the flash device (unlike the operating system swap).

Existing transparent approaches to using flash memory [18, 20, 23] cannot fully exploit flash's performance for two reasons – 1) Accesses to flash happen at a page granularity (4KB), leading to a full page read/write to flash for every access within that page. The write/erase behavior of flash memory often has different expectations on usage, leading to a poor performance. Full pages containing dirty objects have to be written to flash. This behavior leads to write escalation which is bad not only for performance but also for the durability of the flash device. 2) If the application objects are small compared to the page size, only a small fraction of RAM contains useful objects because of caching at a page granularity. Integrating flash as a filesystem cache can increase performance, but the cost/benefit tradeoff of this approach has been questioned before [21].

FlashVM [23] is a system that proposes using flash as a dedicated swap device, that provides hints to the SSD for better garbage collection by batching writes, erases and discards. We propose using 16–32 times more flash than DRAM and in those settings, FlashVM style heuristic batching/aggregating of in-place writes might be of little use purely because of the high write randomness that our targeted applications have. A fully log-structured system would be needed for minimizing erases in such cases. We have built a fully log-structured swap that we use as a comparison point, along with native linux swap, against the SSDAlloc system that works at an object granularity.

Others have proposed redesigning applications to use flash-aware data structures to explicitly handle the asymmetric read/write behavior of flash. Redesigned applications range from databases (BTrees) [19, 24] and Web servers [17] to indexes [6, 8] and key-value stores [7]. Working set objects are cached in RAM more efficiently and the application aggregates objects when writing to flash. While the benefits of this approach can be significant, the costs involved and the extra development effort (requires expertise with the application and flash behavior) are high enough that it may deter most application developers from going this route.

Our goal in this paper is to provide the right set of interfaces (via memory allocators), so that both existing applications and new applications can be easily adapted to use flash. Our approach focuses on exposing flash only via a page based virtual memory interface while internally working at an object level. Similar approach was used in distributed object systems [12], which switched between pages and objects when convenient using custom object handlers. We want to avoid using any custom pointer/handler mechanisms to eliminate intrusive application changes.

Additionally, our approach can improve the cost/benefit ratio of flash-based approaches. If only a few lines of memory allocation code need to be modified to migrate an existing application to a flash-enabled one with performance comparable to that of flash-aware application redesign, this one-time development cost is low compared to the cost of high-density memory. For example, the cost of 1TB of high-density RAM adds roughly \$100K USD to the \$14K base price of the system (e.g., the Dell PowerEdge R910). In comparison, a high-end 320GB SSD sells for \$3200 USD, so roughly 4 servers with 5TB of flash memory cost the same as 1 server with 1 TB of RAM.

SSD Usage Technique	Write Logging	Read/Write < a page	Garbage Collects Dead pages/data	Avoids DRAM Pollution	Persistent Data	High Performance	Programming Ease
SSD Swap							✔
SSD Swap (Write Logged)	✔						✔
SSD `mmap`					✔		✔
Application Rewrite	✔	✔	✔	✔	✔	✔	
SSDAlloc	✔	✔	✔	✔	✔	✔	✔

Table 2: While using SSDs via swap/mmap is simple, they achieve only a fraction of the SSD's performance. Rewriting applications can achieve greater performance but at a high developer cost. SSDAlloc provides simplicity while providing high performance.

SSD Make	reads / sec		writes / sec	
	4KB	0.5KB	4KB	0.5KB
RiDATA (32GB)	3,200	3,700	500	675
Kingston (64GB)	3,300	4,200	1,800	2,000
Intel X25-E (32GB)	26,000	44,000	2,200	2,700
Intel X25-V (40GB)	27,000	46,000	2,400	2,600
Intel X25-M G2 (80GB)	29,000	49,000	2,300	2,500

Table 3: SSDAlloc can take full advantage of object-sized accesses to the SSD, which can often provide significant performance gains over page-sized operations.

3 SSDAlloc's Design

In this section we describe the design of SSDAlloc. We first start with describing the networked systems' requirements from a hybrid DRAM/SSD setting for high-performance and ease of programming. Our high level goals for integrating SSDs into these applications are:

- To present a simple interface such that the applications can be run mostly unmodified – Applications should use the same programming style and interfaces as before (via virtual memory managers), which means that objects, once allocated, always appear to the application at the same locations in the virtual memory.
- To utilize the DRAM in the system as efficiently as possible – Since most of the applications that we focus on allocate large number of objects and operate over them with little locality of reference, the system should be no worse at using DRAM than a custom DRAM based object cache that efficiently packs as many hot objects in DRAM as possible.
- To maximize the SSD's utility – Since the SSD's read performance and especially the write performance suffer with the amount of data transferred, the system should minimize data transfers and (most importantly) avoid random writes.

SSDAlloc employs many clever design decisions and policies to meet our high level goals. In Sections 3.1 and 3.4, we describe our page-based virtual memory system using a modified heap manager in combination with a user-space on-demand page materialization runtime that appears to be a normal virtual memory system to the application. In reality, the virtual memory pages are materialized in an on-demand fashion from the SSD by intercepting page faults. To make this interception as precise as possible, our allocator aligns the application level objects to always start at page boundaries. Such a fine grained interception allows our system to act at an application object granularity and thereby increases the efficiency of reads, writes and garbage collection on the SSD. It also helps in the design of a system that can easily serialize the application's objects to the persistent storage for a subsequent usage.

In Section 3.2, we describe how we use the DRAM efficiently. Since most of the application's objects are smaller than a page, it makes no sense to use all of the DRAM as a page cache. Instead, most of DRAM is filled with an object cache, which packs multiple useful objects per page, and one which is not directly accessible to the application. When the application needs a page, it is dynamically materialized, either from the object cache or from the SSD.

In Sections 3.3 and 3.5 we describe how we manage the SSD as an efficient log-structured object store. In order to reduce the amount of data read/written to the SSD, the system uses the object size information, given to the memory allocator by the application, to transfer only the objects, and not whole pages containing them. Since the objects can be of arbitrary sizes, packing them together and writing them in a log not only reduces the write volume, but also increase the SSD's lifetime.

Table 2 presents an overview of various techniques by which SSDs are used as program memory today and provides a comparison to SSDAlloc by enumerating the high-level goals that each technique satisfies. We now describe our design in detail starting with our virtual address allocation policies.

3.1 SSDAlloc's Virtual Memory Structure

SSDAlloc ideally wants to non-intrusively observe what objects the application reads and writes. The virtual memory (VM) system provides an easy way to detect what pages have been read or written, but there is no easy way to detect at a finer granularity. Performing copy-on-write and comparing the copy with the original can be used for detecting changes, but no easy mechanism de-

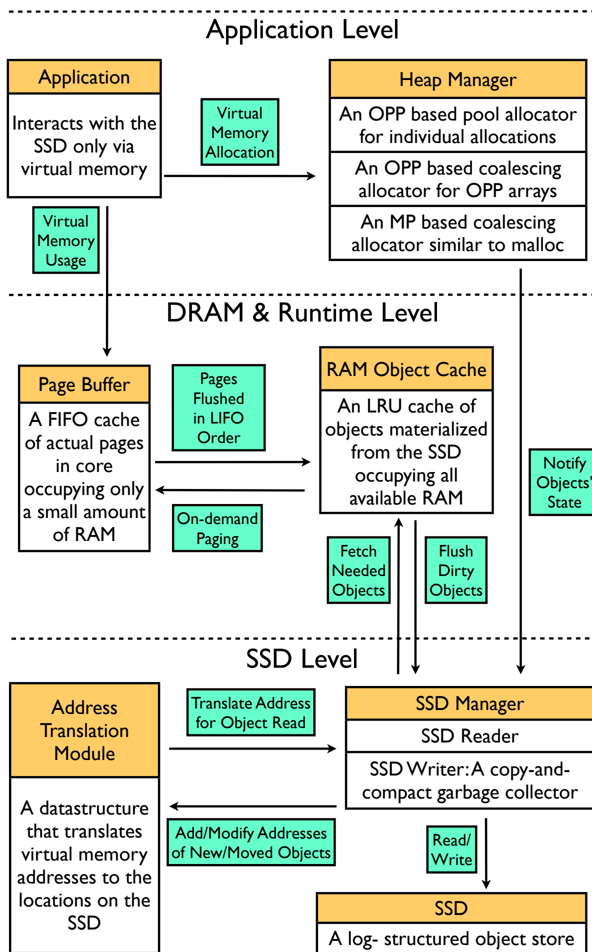


Figure 1: SSDAlloc uses most of RAM as an object-level cache, and materializes/dematerializes pages as needed to satisfy the application's page usage. This approach improves RAM utilization, even though many objects will be spread across a greater range of virtual address space.

termines what parts of a page were read. Instead, SSDAlloc uses the observation that virtual address space is relatively inexpensive compared to actual DRAM, and reorganizes the behavior of memory allocation to use the VM system to observe object behavior. Servers typically expose 48 bit address spaces (256TB) while supporting less than 1TB of physical RAM, so virtual addresses are at least 256x more plentiful.

We propose the Object Per Page (OPP) model, using which, if an application requests memory for an object, the object is placed on its own page of virtual memory, yielding a single page for small objects, or more (contiguous) when the object exceeds the page size. The object is always placed at the start of the page and the rest of the page is not utilized for memory allocation. In reality, however, we employ various optimizations (described in Section 3.2) to eliminate the physical memory wastage that can occur because of such a lavish virtual memory usage. An OPP memory manager can be implemented just by maintaining a pool of pages (details of the actual memory manager used are given in Section 3.4). OPP is suitable for individual object allocations, typical of the applications we focus on. OPP objects are stored on the SSD in a log-structured manner (details are explained in Section 3.5). Additionally, using virtual memory based page-usage information, we can accurately determine which objects are being read and written (since there is only one object per page). However, it is not straightforward to use arrays of objects in this manner. In an OPP array, each object is separated by the page's size as opposed to the object's size. While it is possible to allocate OPP arrays in such a manner, it would require some code modifications to be able to use arrays in which objects separated by page boundaries as opposed being separated by object boundaries. We describe later in Section 3.4 how an OPP based coalescing allocator can be used to allocate OPP based arrays.

3.1.1 Contiguous Array Allocations

In the C programming language, array allocations via `malloc`/`calloc` expect array elements to be contiguous. We present an option called Memory Pages (MP) which can do this. In MP, when the application asks for a certain amount of memory, SSDAlloc returns a pointer to a region of virtual address space with the size requested. We use a `ptmalloc` [5] style coalescing memory manager (further explained in Section 3.4) built on top of bulk allocated virtual memory pages (via `brk`) to obtain a system which can allocate C style arrays. Internally, however, the pages in this space are treated like page sized OPP objects. For the rest of the paper, we treat MP pages as page sized OPP objects.

While the design of OPP efficiently leverages the virtual memory system's page level usage information to determine application object behavior, it could lead to DRAM space wastage because the rest of the page beyond the object would not be used. To eliminate this wastage, we organize the physical memory such that only a small portion of DRAM contains actual materializations of OPP pages (Page Buffer) while the rest of the available DRAM is used as a compact hot object cache.

3.2 SSDAlloc's Physical Memory Structure

The SSDAlloc runtime system eases application transparency by allowing objects to maintain the same virtual address over their lifetimes, while their physical location may be in a temporarily-materialized physical page mapped to its virtual memory page in the Page Buffer, the RAM Object Cache, or the SSD. Not only does the runtime materialize physical pages as needed, but it also

reclaims them when their usage drops. We first describe how objects are cached compactly in DRAM.

RAM Object Cache – Objects are cached in *RAM object cache* in a compact manner. RAM object cache occupies available portion of DRAM while only a small part of DRAM is use for pages that are currently in use (shown in Figure 1). This decision provides several benefits – 1) Objects cached in RAM can be accessed much faster than the SSD, 2) By performing usage-based caching of objects instead of pages, the relatively small RAM can cache more useful objects when using OPP, and 3) Given the density trends of SSD and RAM, object caching is likely to continue being a useful optimization going forward.

RAM object cache is maintained in LRU fashion. It indexes objects using their virtual memory page address as the key. An OPP object in RAM object cache is indexed by its OPP page address, while an MP page (a 4KB OPP object) is indexed with its MP page address. In our implementation, we used a hashtable with the page address as the key for this purpose. Clean objects being evicted from the RAM object cache are deallocated while dirty objects being evicted are enqueued to the SSD writer mechanism (shown in Figure 1).

Page Buffer – Temporarily materialized pages (in physical memory) are are collectively known as the Page Buffer. These pages are materialized in an on-demand fashion (described below). Page Buffer size is application configurable, but in most of the applications we tested, we found that a Page Buffer of size less than 25MB was sufficient to bring down the rate of page materializations per second to the throughput of the application. However, regardless of the size of the Page Buffer, physical memory wastage from using OPP has to be minimized. To minimize this wastage we make the rest of the active OPP physical page (portion beyond the object) a part of the RAM object cache. RAM object cache is implemented such that the shards of pages that materialize into physical memory are used for caching objects.

SSDAlloc's Paging – For a simple user space implementation we implement the Page Buffer via memory protection. All virtual memory allocated using SSDAlloc is protected (via `mprotect`). A page usage is detected when the protection mechanism triggers a fault. The required page is then unprotected (only read or write access is given depending on the type of fault to be able to detect writes separately) and its data is then populated in the seg-fault handler – an OPP page is populated by fetching the object from RAM object cache or the SSD and placing it at the front of the page. An MP page is populated with a copy of the page (a page sized object) from RAM object cache or the SSD.

Pages dematerialized from Page Buffer are converted to objects. Those objects are pushed into the RAM object cache, the page is then `madvise`d to be not needed and finally, the page is reprotected (via `mprotect`) – in case of OPP/MP the object/page is marked as dirty if the page faults on a write.

Page Buffer can be managed in many ways, with the simplest way being FIFO. Page Buffer pages are unprotected, so our user space implementation based runtime would have no information about how a page would be used while it remains in the Page Buffer, making LRU difficult to implement. For simplicity, we used FIFO in our current implementation. The only penalty is that if a dematerialized page is needed again then the page has to be rematerialized from RAM.

OPP can have more virtual memory usage than `malloc` for the same amount of data allocated. While MP will round each virtual address allocation to the next highest page size, the OPP model allocates one object per page. For 48-bit address spaces, the total number of pages is 2^{36} ($\approx$ 64 Billion objects via OPP). For 32-bit systems, the corresponding number is 2^{20} ($\approx$ 1 million objects). Programs that need to allocate more objects on 32-bit systems can use MP instead of OPP. Furthermore, SSDAlloc can coexist with standard `malloc`, so address space usage can be tuned by moving only necessary allocations to OPP.

While the separation between virtual memory and physical memory presents many avenues for DRAM optimization, it does not directly optimize SSD usage. We next present our SSD organization.

3.3 SSDAlloc's SSD Maintenance

To overcome the limitations on random write behavior with SSDs, SSDAlloc writes the dirty objects when flushing the RAM object cache to the SSD in a log-structured [22] manner. This means that the objects have no fixed storage location on the SSD – similar to flash-based filesystems [11]. We first describe how we manage the mapping between fixed virtual address spaces to ever-changing log-structured SSD locations. Our SSD writer/garbage-collector is described later.

To locate objects on the SSD, SSDAlloc uses a data structure called the **Object Table**. While the virtual memory addresses of the objects are their fixed locations, Object Tables store their ever-changing SSD locations. Object Tables are similar to page tables in traditional virtual memory systems. Each Object Table has a unique identifier called the OTID and it contains an array of integers representing the SSD locations of the objects it indexes. An object's Object Table Offset (OTO) is the offset in this array where its SSD location is stored. The 2-tuple <OTID, OTO> is the object's internal persistent pointer.

To efficiently fetch the objects from the SSD when they are not cached in RAM, we keep a mapping between

each virtual address range (as allocated by the OPP or the MP memory manager) in use by the application and its corresponding Object Table, called an **Address Translation Module** (ATM). When the object of a page that is requested for materialization is not present in the RAM object cache, <OTID,OTO> of that object is determined from the page's address via an ATM lookup (shown in Figure 1). Once the <OTID,OTO> is known, the object is fetched from the SSD, inserted into RAM object cache and the page is then materialized. The ATM is only used when the RAM object cache does not have the required objects. A successful lookup results in a materialized physical page that can be used without runtime system intervention for as long as the page resides in the Page Buffer. If the page that is requested does not belong to any allocated range, then the segmentation fault is a program error. In that case the control is returned to the originally installed seg-fault handler.

The ATM indexes and stores the 2-tuples <Virtual Memory Range, OTID> such that when it is queried with a virtual memory page address, it responds with the <OTID,OTO> of the object belonging to the page. In our implementation, we chose a balanced binary search tree for various reasons – 1) virtual memory range can be used as a key while the OTID can be used as a value. The search tree can be queried using an arbitrary page address and by using a binary search, one can determine the virtual memory range it belongs to. Using the queried page's offset into this range, the relevant object's OTO is determined, 2) it allows the virtual memory ranges to be of any size and 3) it provides a simple mechanism by which we can improve the lookup performance – by reducing the number of Object Tables, there by reducing the number of entries in the binary search tree. Our heap manager which allocates virtual memory (in OPP or MP style) always tries to keep the number of virtual memory ranges in use to a minimum to reduce the number of Object Tables in use. Before we describe our heap manager design, we present a few simple optimizations to reduce the size of Object Tables.

We try to store the Object Tables fully in DRAM to minimize multiple SSD accesses to read an object. We perform two important optimizations to reduce the size overhead from the Object Tables. First – to be able to index large SSDs for arbitrarily sized objects, one would need a 64 bit offset that would increase the DRAM overhead for storing Object Tables. Instead, we store a 32 bit offset to an aligned 512 byte SSD sector that contains the start of the object. While objects may cross the 512 byte sector boundaries, the first two bytes in each sector are used to store the offset to the start of the first object starting in that sector. Each object's on-SSD metadata contains its size, using which, we can then find the rest of the object boundaries in that sector. We can index 2TB of SSD this way. 40 bit offsets can be used for larger SSDs.

Our second optimization addresses Object Table overhead from small objects. For example, four byte objects can create 100% DRAM overhead from their Object Table offsets. To reduce this overhead, we introduce object batching – small objects are batched into larger contiguous objects. We batch enough objects together such that the size of the larger object is at least 128 bytes (restricting the Object Table overhead to a small fraction – $\frac{1}{32}$). Pages, however, are materialized in regular OPP style – one small object per page. However, batched objects are internally maintained as a single object.

3.4 SSDAlloc's Heap Manager

Internally, SSDAlloc's virtual memory allocation mechanism works like a memory manager over large Object Table allocations (shown in Figure 1). This ensures that a new Object Table is not created for every memory allocation. The Object Tables and their corresponding virtual memory ranges are created in bulk and memory managers allocate from these regions to increase ATM lookup efficiency. We provide two kinds of memory managers – An object pool allocator which is used for individual allocations and a `ptmalloc` style coalescing memory manager. We keep the pool allocator separate from the coalescing allocator for the following reasons: 1) Many of our focus applications prefer pool allocators, so providing a pool allocator further eases their development, 2) Pool allocators reduce the number of page reads/writes by not requiring coalescing, and 3) Pool allocators can export simpler memory usage information, increasing garbage collector efficiency.

Object Pool Allocator: SSDAlloc provides an object pool allocator for allocating objects individually via OPP. Unlike traditional pool allocators, we do not create pools for each object type, but instead create pools of different size ranges. For example, all objects of size less than 0.5KB are allocated from one pool, while objects with sizes between 0.5KB and 1KB are allocated from another pool. Such pools exist for every 0.5KB size range, since OPP performs virtual memory operations at page granularity. Despite the pools using size ranges, we avoid wasting space by obtaining the actual object size from the application at allocation time, and using this size both when the object is stored in the RAM object cache, and when the object is written to the SSD. When reading an object from the SSD, the read is rounded to the pool size to avoid multiple small reads.

SSDAlloc maintains each pool as a free list – a pool starts with a single allocation of 128 objects (one Object Table, with pages contiguous in virtual address space) initially and doubles in size when it runs out of space (with a single Object Table and a contiguous virtual memory range). No space in the RAM object cache or

the SSD is actually used when the size of pool is increased, since only virtual address space is allocated. The pool stops doubling in size when it reaches a size of 10,000 (configurable) and starts linearly increasing in steps of 10,000 from then on. The free-list state of an object can be used to determine if an object on the SSD is garbage, enabling object-granularity garbage collection. This type of a separation of the heap-manager state from where the data is actually stored is similar to the "frame-heap" implementation of Xerox Parc's Mesa and Cedar languages [15].

Like Object Tables, we try to maintain free-lists in DRAM, so the free list size is tied to the number of free objects, instead of the total number of objects. To reduce the size of the free list we do the following: the free list actively indexes the state of only one Object Table of each pool at any point of time, while the allocation state for the rest of the Object Tables in each pool is managed using a compact bitmap notation along with a count of free objects in each Object Table. When the heap manager cannot allocate from the current one, it simply changes the current Object Table's free list representation to a bitmap and moves on to the Object Table with the largest number of free objects, or it increases the size of the pool.

Coalescing Allocator: SSDAlloc's coalescing memory manager works by using memory managers like ptmalloc [5] over large address spaces that have been reserved. In our implementation we use a simple *best-first with coalescing* memory manager [5] over large pre-allocated address spaces, in steps of 10,000 (configurable) pages; no DRAM or SSD space is used for these pre-allocations, since only virtual address space is reserved. Each object/page allocated as part of the coalescing memory manager is given extra metadata space in the header of a page to hold the memory manager information (objects are then appropriately offset). OPP arrays of any size can be allocated by performing coalescing at the page granularity, since OPP arrays are simply arrays of pages. MP pages are treated like pages in the traditional virtual memory system. The memory manager works exactly like traditional `malloc`, coalescing freely at byte granularity. Thus, MP with our *Coalescing Allocator* can be used as a drop-in replacement for log-structured swap.

A dirty object evicted by RAM object cache needs to be written to the SSD's log and the new location has to be entered at its OTO. This means that the older location of the object has to be garbage collected. An OPP object on the SSD which is in a free-list also needs to be garbage-collected. Since SSDs do not have the mechanical delays associated with a moving disk head, we can use a simpler garbage collector than the seek-optimized ones developed for disk-based log-structured file systems [22]. Our cleaner performs a "read-modify-write" operation over the SSD sequentially – it reads any live objects at the head of the log, packs them together, and writes them along with flushed dirty objects from RAM.

3.5 SSDAlloc's Garbage Collector

The SSDAlloc Garbage Collector (GC) activates whenever the RAM object cache has evicted enough number of dirty objects (as shown in Figure 1) to amortize the cost of writing to the SSD. We use a simple read-modify-write garbage collector, which reads enough partially-filled blocks (of configurable size, preferably large) at the head of the log to make space for the new writes. Each object on the SSD has its 2-tuple <OTID,OTO> and its size as the metadata, used to update the Object Table. This back pointer is also used to figure out if the object is garbage, by matching the location in the Object Table with the actual offset. To minimize the number of reads per iteration of the GC on the SSD, we maintain in RAM the amount of free space per 128KB block. These numbers can be updated whenever an object in an erase block is moved elsewhere (live object migration for compaction), when a new object is written to it (for writing out dirty objects) or when the object is moved to a free-list (object is "free").

While the design so far focused on obtaining high performance from DRAM and flash in a hybrid setting, memory allocated via SSDAlloc is not non-volatile. We now present our durability framework to preserve application memory and state on the SSD.

3.6 SSDAlloc's Durability Framework

SSDAlloc helps applications make their data persistent across reboots. Since SSDAlloc is designed to use much more SSD-backed memory than the RAM in the system, the runtime is expected to maintain the data persistent across reboots to avoid the loss of work.

SSDAlloc's checkpointing is a way to cleanly shutdown an SSDAlloc based application while making objects and metadata persistent to be used across reboots. Objects can be made persistent by simply flushing all the dirty objects from RAM object cache to the SSD. The state of the heap-manager, however, needs more support to be made persistent. The bitmap style free list representation of the OPP pool allocator makes the heap-manager representation of individually allocated OPP objects easy to be serialized to the SSD. However, the heap-manager information as stored by a coalescing memory manager used by the OPP based array allocator and the MP based memory allocator would need a full scan of the data on the SSD to be regenerated after a reboot. Our current implementation provides durability only for the individually allocated OPP objects and we wish to provide durability for other types of SSDAlloc data in the future.

We provide durability for the heap-manager state of the individually allocated OPP objects by reserving a

known portion of the SSD for storing the corresponding Object Tables and the free list state (a bitmap). Since the maximum Object Table space to object size overhead ratio is $\frac{1}{32}$, we reserve slightly more than $\frac{1}{32}$ of the total SSD space (by using a file that occupies that much space) where the Object Tables and the free list state can be serialized for later use.

It should be possible to garbage collect dead objects across reboots. This is handled by making sure that our copy-and-compact garbage collector is always aware of all the OTIDs that are currently active within the SSDAlloc system. Any object with an unknown OTID is garbage collected. Additionally, any object with an OTID that is active is garbage collected only according to the criteria discussed in Section 3.5.

Virtual memory address ranges of each Object Table must be maintained across reboots, because checkpointed data might contain pointers to other checkpointed data. We store the virtual memory address range of each Object Table in the first object that this Object Table indexes. This object is written once at the time of creation of the Object Table and is not made available to the heap manager for allocation.

3.7 SSDAlloc's Overhead

We observe that the overhead introduced by the SSDAlloc's runtime mechanism is minor compared to the performance limits of today's high-end SSDs. On a test machine with a 2.4 GHz quad-core processor, we benchmark the SSDAlloc's runtime mechanism to arrive at that conclusion. To benchmark the latency overhead of the signal handling mechanism, we protect 200 Million pages and then measure the maximum seg-fault generation rate that can be attained. For measuring the the ATM lookup latency, we build an ATM with a million entries and then measure the maximum lookup throughput that can be obtained. To benchmark the latency of an on-demand page materialization of an object from the RAM object cache to a page within the Page Buffer, we populate a page with random data and measure the latency. To benchmark the page dematerialization of a page from the Page Buffer to an object in the RAM object cache, we copy the contents of the page elsewhere, `madvise` the page as not needed and reprotect the page using `mprotect` and measure the total latency. To benchmark the latency of TLB misses (through L3) we use a CPU benchmarking tool, the Calibrator [2], by allocating 15GB of memory per core. Table 4 presents the results. Latencies of all the overheads clearly indicate that they would not be a bottleneck even for the high-end SSDs like the FusionIO IOXtreme drives, which can provide up to 250,000 IOPS. In fact, one would need 5 such SSDs for the SSDAlloc runtime to saturate the CPU.

The largest CPU overhead is from the signal handling mechanism, which is present only because of a user space implementation. With an in kernel implementation, the VM pager can be used to manage the Page Buffer, which would further reduce the CPU usage. We designed OPP for applications with high read randomness without much locality, because of which, using OPP will not greatly increase the number of TLB (through L3) misses. Hence, applications that are not bottlenecked by DRAM (but by CPU, network, storage capacity, power consumption or magnetic disk) can replace DRAM with high-end SSDs via SSDAlloc and reduce hardware expenditure and power costs. For example, Facebook's memcache servers are bottlenecked by network parameters [3]; their peak performance of 200,000 tps per server can be easily obtained by using today's high-end SSDs as RAM extension via SSDAlloc.

Overhead Source	Avg. Latency (μsec)
TLB Miss (DRAM read)	0.014
ATM Lookups	0.046
Page Materialization	0.138
Page Dematerialization	0.172
Signal Handling	0.666
Combined Overhead	0.833

Table 4: SSDAlloc's overheads are quite low, and place an upper limit of over 1 million operations per second using low-end server hardware. This request rate is much higher than even the higher-performance SSDs available today, and is higher than even what most server applications need from RAM.

DRAM overhead created from the Object Tables is compensated by the performance gains. For example, a 300GB SSD would need 10GB and 300MB of space for Object Tables when using OPP and MP respectively for creating 128 byte objects. However, SSDAlloc's random read/write performance when using OPP is 3.5 times better than when using MP (shown in Section 5). Additionally, for the same random write workload OPP generates 32 times less write traffic to the SSD when compared to MP and thereby increases the lifetime of the SSD. Additionally, with an in kernel implementation, either the page tables or the Object Tables will be used as they both serve the same purpose, further reducing the overhead of having the Object Tables in DRAM.

4 Implementation and the API

We have implemented our SSDAlloc prototype as a C++ library in roughly 10,000 lines of code. It currently supports SSD as the only form of flash memory, though it could later be expanded, if necessary, to support other forms of flash memory. In our current implementation, applications can coexist by creating multiple files on the SSD. Alternatively, an application can use the entire SSD, as a raw disk device for high performance. While the current implementation uses flash memory via an I/O

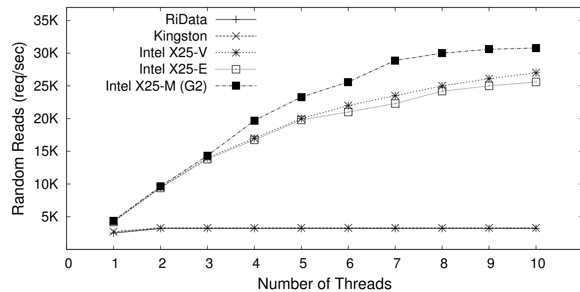


Figure 2: SSDAlloc's thread-safe memory allocators allow applications to exploit the full parallelism of many SSDs, which can yield significant performance advantages. Shown here is the performance for 4KB reads.

controller such an overhead may be avoided in the future [13]. We present an overview of the implementation via a description of the API.

ssd_oalloc: *void* ssd_oalloc(int numObjects, int objectSize)*: is used for OPP allocations – both individual and array allocations. If *numObjects* is 1 then the object is allocated from the in-built OPP pool allocator. If it is more than 1, it is allocated from the OPP coalescing memory manager.

ssd_malloc: *void* ssd_malloc(size_t size)*: allocates *size* bytes of memory using the heap manager (described in Section 3.4) on MP pages. Similar calls exist for **ssd_calloc** and **ssd_realloc**.

ssd_free: *void ssd_free(void* va_address)*: deallocates the objects whose virtual allocation address is *va_address*. If the allocation was via the pool allocator then the <OTID,OTO> of the object is added to the appropriate free list. In case of array allocations, the inbuilt memory manager frees the data according to our heap manager. SSDAlloc is designed to work with low level programming languages like 'C'. Hence, the onus of avoiding memory leaks and of freeing the data appropriately is on the application.

checkpoint: *int checkpoint(char* filename)*: flushes all dirty objects to the SSD and writes all the Object Tables and free-lists of the application to the file *filename*. This call is used to make the objects of an application durable.

restore: *int restore(char* filename)* : It restores the SSDAlloc state for the calling application. It reads the file (*filename*) containing the Object Tables and the free list state needed by the application and `mmap`s the necessary address for each Object Table (using the first object entry) and then inserts the mappings into the ATM as described in Section 3.6.

SSDs scale performance with parallelism. Figure 2 shows how some high-end SSDs have internal parallelism (for 0.5KB reads, other read sizes also have parallelism). Additionally, multiple SSDs could be used with in an application. All SSDAlloc functions, including the heap manager, are implemented in a thread safe manner to be able to exploit the parallelism.

4.1 Migration to SSDAlloc

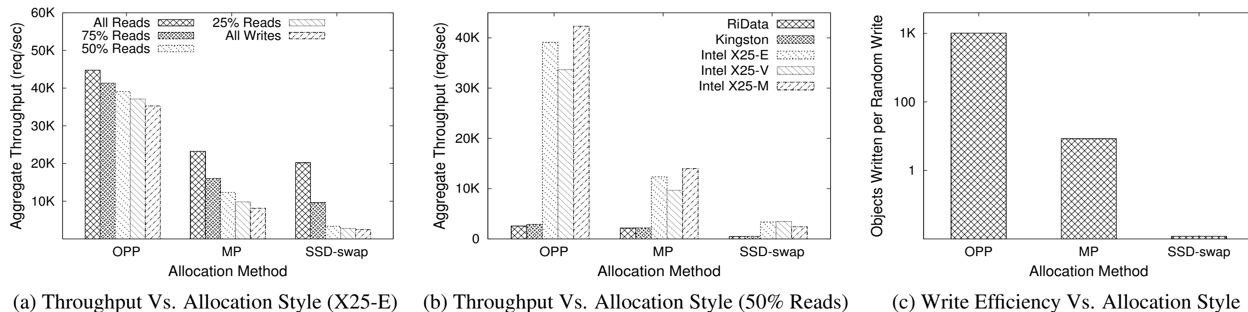
We believe that SSDAlloc is suited to the memory-intensive portions of server applications with minimal to no locality of reference, and that migration should not be difficult in most cases – our experience suggests that only a small number of data types are responsible for most of the memory usage in these applications. The following scenarios of migration are possible for such applications to embrace SSDAlloc:

- Replace all calls to `malloc` with `ssd_malloc`: Application would then use the SSD as a log-structured page store and use the DRAM as a page cache. Application's performance would be better than when using the SSD via unmodified Linux swap because it would avoid random writes and circumvent other legacy swap system overheads that are more clearly quantified in FlashVM [23].
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_malloc`: Application can then avoid SSDAlloc's runtime intervention (copying data between Page Buffer and RAM object cache) for non-memory intensive datastructures and can thereby slightly reduce its CPU utilization.
- Replace all `malloc` calls made to allocate memory intensive datastructures of the application with `ssd_oalloc`: Application would then use the SSD as a log-structured object store only for memory intensive objects. Application's performance would be better than when using the SSD as a log-structured swap because now the DRAM and the SSD would be managed at an object granularity.

In our evaluation of SSDAlloc, we tested all the above migration scenarios to estimate the methodology that provides the maximum benefit for applications in a hybrid DRAM/SSD setting.

5 Evaluation Results

In this section we evaluate SSDAlloc using microbenchmarks and applications built or modified to use SSDAlloc. We first present microbenchmarks to test the limits of benefits from using SSDAlloc versus SSD-swap. We also examine the performance of memcached (with SSDAlloc and SSD-swap), a popular key-value store used in datacenters, where SSDs have been shown to minimize energy consumption [7]. Later, we benchmark a B+Tree index for SSDs, where we replace all calls to `malloc` with `ssd_malloc` to see the benefits and impact of an automated migration to SSDAlloc.



(a) Throughput Vs. Allocation Style (X25-E) (b) Throughput Vs. Allocation Style (50% Reads) (c) Write Efficiency Vs. Allocation Style

Figure 3: Microbenchmark results on 32GB object (128 byte each) array. In (a), OPP works best (1.8–3.5 times over MP and 2.2–14.5 times over swap), MP and swap take a huge performance hit when write traffic increases. In (b), OPP, on all SSDs, trumps all other methods by reducing read and write traffic. In (c), OPP has the maximum write efficiency (31.5 times over MP and 1013 times over swap) by writing only dirty objects as opposed to writing full pages containing them.

After that, we compare the performance of systems designed to use SSDAlloc to the same system specifically customized to use the SSD directly, to evaluate the overhead from SSDAlloc's runtime. We examine a network packet cache backend that was built using transparent SSDAlloc techniques described in this paper and also the non-transparent mechanism described in our workshop paper [8]. We also evaluate the performance of a web proxy/WAN accelerator cache index for SSDs introduced in prior work [9, 8] and similar to the problems addressed more recently [6, 14]. Here, we demonstrate how using OPP makes efficient use of DRAM while providing high performance.

In all these experiments we evaluate applications using three different allocation methods: **SSD-swap** (via `malloc`), **MP** or log-structured SSD-swap (via `ssd_malloc`), **OPP** (via `ssd_oalloc`). Our evaluations use five kinds of SSDs and two types of servers. The SSDs and some of their performance characteristics are shown in Table 3. The two servers we use have a single core 2GHz CPU with 4GB of RAM and a quad-core 2.4GHz CPU with 16GB of RAM respectively.

5.1 Microbenchmarks

We examine the performance of random reads and writes in an SSD-augmented memory by accessing a large array of 128 byte objects – an array of total size of 32GB using various SSDs. We further restrict the accessible RAM in the system to 1.5GB to test out-of-DRAM performance. We access objects randomly (read or write) 2 million times per test. The array is allocated using four different methods – SSD-swap (via `malloc`), MP (via `ssd_malloc`), OPP (via `ssd_oalloc`). Object Tables for each of OPP, and MP occupy 1.1GB and 34MB respectively. Page Buffers are restricted to a size of 25 MB (it was sufficient to pin a page down while it was being accessed in an iteration). Remaining memory was used by the RAM object cache. To exploit the SSD's parallelism, we run 8–10 threads that perform the random accesses in parallel.

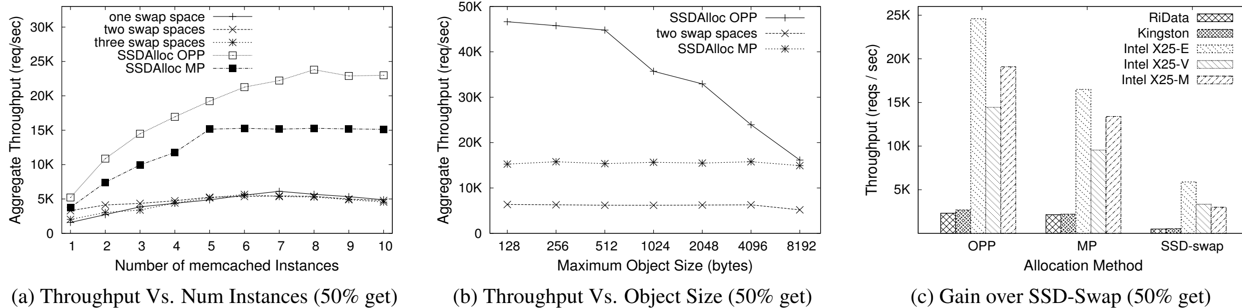
	OPP	MP	SSD-swap
Average (μsec)	257	468	624
Std Dev (μsec)	66	98	287

Table 5: Response times show that OPP performs best, since it can make the best use of the block-level performance of the SSD whereas MP provides page-level performance. SSD-swap performs poorly due to worse write behavior.

The results of this microbenchmark are shown in Figure 3. Figure 3(a) shows how (for the Intel X25-E SSD) allocating objects via OPP achieves much higher performance. OPP beats MP by a factor of **1.8–3.5** times depending on the write percentage and it beats SSD-swap by a factor of **2.2–14.5** times. As the write traffic increases, MP and SSD-swap fare poorly due to reading/writing at a page granularity. OPP reads only 512 byte sector per object access as opposed to reading a 4KB page; it dirties only 128 bytes as opposed to dirtying 4KB per random write.

Figure 3(b) demonstrates how OPP performs better than all the allocation methods across all the SSDs when 50% of the operations are writes. OPP beats MP by a factor of **1.4–3.5** times and it beats SSD-swap by a factor of **5.5–17.4** times. Table 5 presents response time statistics when using the Intel X25-E SSD. OPP has the lowest averages and standard deviations. SSD-swap has a high average response time compared to OPP and MP. This is mainly because of storage sub-system inefficiencies and random writes (quantified more clearly in [23]).

Figure 3(c) quantifies the write optimization obtained by using OPP in log scale. OPP writes at an object granularity, which means that it can fit more number of dirty objects in a given write buffer when compared to MP. When a 128KB write buffer is used, OPP can fit nearly 1024 dirty objects in the write buffer while MP can fit only around 32 pages containing dirty objects. Hence, OPP writes more number of dirty objects to the SSD per random write when compared to both MP and SSD-



(a) Throughput Vs. Num Instances (50% get) (b) Throughput Vs. Object Size (50% get) (c) Gain over SSD-Swap (50% get)

Figure 4: Memcached results. In (a), OPP outperforms MP and SSD-swap by factors of 1.6 and 5.1 respectively (mix of 4byte to 4KB objects). In (b), SSDAlloc's use of objects internally can yield dramatic benefits, especially for smaller memcached objects. In (c), SSDAlloc beats SSD-Swap by a factor of 4.1 to 6.4 for memcached tests (mix of 4byte to 4KB objects).

swap (which makes a random write for every dirty object). OPP writes **1013** times more efficiently compared to SSD-swap and **31.5** times compared to MP (factors independent of SSD make). Additionally, OPP not only increases write efficiency but also writes **31.5** times less data compared to MP and SSD-swap for the same workload by working at an object granularity and thereby increases the SSD lifetime by the same factor.

Overall, OPP trumps SSD-swap by huge gain factors. It also outperforms MP by large factors providing a good insight into the benefits that OPP would provide over log-structured swaps. Such benefits scale inversely with the size of the object. For example with 1KB objects OPP beats MP by a factor of **1.6–2.8** and with 2KB objects the factor is **1.4–2.3**.

5.2 Memcached Benchmarks

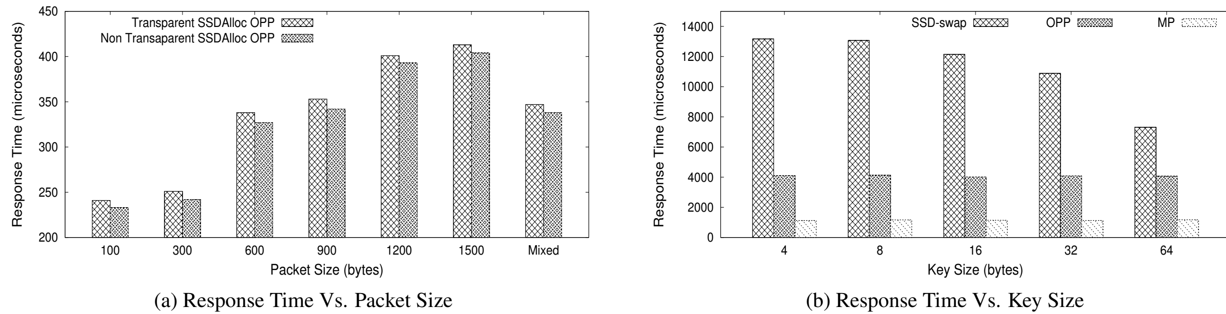
To demonstrate the simplicity of SSDAlloc and its performance benefits for existing applications, we modify memcached. Memcached uses a custom slab allocator to allocate values and regular `malloc`s for keys. We replaced memcache's slabs with OPP (`ssd_oalloc`) and with MP(`ssd_malloc`) to obtain two different versions. These changes require modifying 21 lines of code out of over 11,000 lines in the program. When using MP, we replaced `malloc` with `ssd_malloc` inside memcache's slab allocator (used only for allocating values).

We compare these versions with an unmodified memcached using SSD-swap. For SSDs with parallelism we create multiple swap partitions on the same SSD. We also run multiple instances of memcached to exploit CPU and SSD parallelism. Figure 4 shows the results.

Figure 4(a) shows the aggregate throughput obtained using a 32GB Intel X25-E SSD (2.5GB RAM), while varying the number of memcached instances used. We compare five different configurations – memcached with OPP and MP, memcached with one, two and three swap partitions on the same SSD. For this experiment we populate memcached instances with object sizes distributed uniformly randomly from 4 bytes to 4KB such that the total size of objects inserted is 30GB. For benchmarking, we generate 1 million memcached *get* and *set* requests (100% hitrate) each using four client machines that statically partition the keys and distribute their requests to all running memcached instances.

Results indicate that SSDAlloc's write aggregation is able to exploit the device's parallelism, while SSD-swap based memcached is restricted in performance, mainly due to the swap's random write behavior. OPP (at 8 instances of memcached) beats MP (at 6 instances of memcached) and SSD-swap (at 6 instances of memcached on two swap partitions) by factors of 1.6 and 5.1 respectively by working at an object granularity, for a mix of object sizes from 4bytes to 4KB. While using SSD-Swap with two partitions lowers the standard deviation of the response time, SSD-Swap had much higher variance in general. For SSD-Swap, the average response time was 667 microseconds and the standard deviation was 398 microseconds, as opposed to OPP's response times of 287 microseconds with a 112 microsecond standard deviation (high variance due to synchronous GC).

Figure 4(b) shows how object size determines memcached performance with and without OPP (Intel X25-E SSD). Here, we generate requests over the entire workload without much locality. We compare the aggregate throughput obtained while varying the maximum object size (actual sizes are distributed uniformly from 128 bytes to limit). We perform this experiment for three settings – 1) Eight memcached instances with OPP, 2) Six memcached instances with MP and 3) Six memcached instances with two swap partitions. We picked the number of instances from the best performing numbers obtained from the previous experiment. We notice that as the object size decreases, memcached with OPP performs much better than when compared to memcached with SSD-swap and MP. This is due to the fact that using OPP moves objects to/from the SSD, instead of pages, resulting in smaller reads and writes. The slight drop in performance in case of MP and SSD-swap when moving from 4KB object size limit to 8KB is because the runtime



(a) Response Time Vs. Packet Size

(b) Response Time Vs. Key Size

Figure 5: Packet Cache Benchmarks: In (a) we see that SSDAlloc's runtime mechanism adds only up to 20 microseconds of latency overhead, while there was no significant difference in throughput. B+Tree Benchmarks: In (b), we see that SSDAlloc's ability to internally use objects beats page-sized operations of MP or SSD-swap.

sometimes issues two reads for objects larger than 4KB. When the Object Table indicates that they are contiguous on SSD, we can fetch them together. In comparison, SSD-swap prefetches when possible.

Figure 4(c) quantifies these gains for various SSDs (objects between 4byte and 4KB) at a high insert rate of 50%. The benefits of OPP can be anywhere between **4.1–6.4** times higher than SSD-swap and **1.2–1.5** times higher than MP (log-structured swap). For smaller objects (each 0.5KB) the gains are **1.3–3.2** and **4.9–16.4** times respectively over MP and SSD-swap (16.4 factor improvement is achieved on the Intel X25-V SSD). Also, depending on object size distribution, OPP writes anywhere between **3.88–31.6** times more efficiently when compared to MP and **24.71–1007** times compared to SSD-swap (objects written per SSD write). The total write traffic of OPP is also between **3.88–31.6** times less when compared to MP and SSD-swap, increasing the lifetime and reliability of the SSD.

5.3 Packet Cache Benchmarks

Packet caches (and chunk caches) built using SSDs scale the performance of network accelerators [6] and inline data deduplicators [14] by exploiting good random read performance and large capacity of flash. Similar capacity DRAM-only systems will cost much more and also consume more power. We built a packet cache backend that indexes a packet with the SHA1 hash of its contents (using a hash table). We built it via two methods – 1) packets are allocated via OPP (`ssd_oalloc`), and 2) packets are allocated via the non-transparent object get/put based SSDAlloc that we describe in our workshop paper [8] – where the SSD is used directly without any runtime intervention. Remaining data structures in both the systems are allocated via `malloc`. We compare these two implementations to estimate the overhead from SSDAlloc's runtime mechanism for each packet accessed.

For the comparison, we test the response times of packet get/put operations into the backend. We consider many settings – we vary the size of the packet from 100 to 1500 bytes and in another setting we consider a mix of packet sizes (uniformly, from 100 to 1500 bytes). We use a 20 byte SHA1 hash of the packet as the key that is stored in the hashtable (in DRAM) against the packet as the value (on SSD) – the cache is managed in LRU fashion. We generate random packet content from "/dev/random". We use the Intel X25-M SSD and the high-end CPU machine for these experiments, with eight threads for exploiting device parallelism. We first fill the SSD with 32GB worth of packets and then perform 2 million lookups and inserts (after evicting older packets in LRU fashion). In this benchmark, we configured the Page Buffer to hold only a handful of packets such that every page get/put request leads to a signal raise, and an ATM lookup followed by an OPP page materialization.

Figure 5(a) compares the response times of OPP method using the transparent techniques described in this paper and the non-transparent calls described in the workshop paper [8]. The results indicate that the overhead from SSDAlloc's runtime mechanism is only on the order of ten microseconds, there is no significant difference in throughput. Highest overhead observed was for 100 byte packets, where transparent SSDAlloc consumed 6.5% more CPU than the custom SSD usage approach when running at 38K 100 byte packets per second (30.4 Mbps). We believe this overhead is acceptable given the ease of development. We also built the packet cache by allocating packets via MP (`ssd_malloc`) and SSD-swap (`malloc`). We find that OPP based packet cache performed **1.3–2.3** times better than an MP based one and **4.8–10.1** times better than SSD-swap for mixed packets (from 100 to 1500 bytes) across all SSDs. Write efficiency of OPP scaled according to the packet size as opposed to MP and SSD-swap which always write a full page (either for writing a new packet or for editing the heap manager data by calling `ssd_free` or `free`). Using an OPP packet cache, three Intel SSDs can accelerate a 1Gbps link (1500 byte packets at 100% hit rate). Whereas, MP and SSD-swap would need 5 and 12 SSDs respectively.

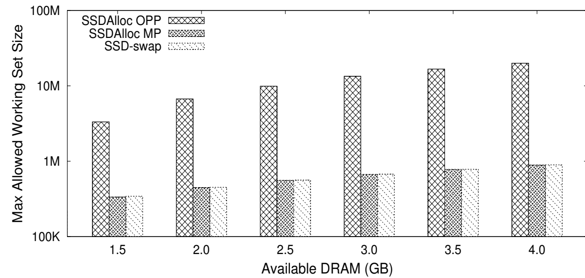


Figure 6: HashCache benchmarks: SSDAlloc OPP option can beat MP and SSD-Swap on RAM requirements due to caching objects instead of pages. The maximum size of a completely random working set of index entries each allocation method can cache in DRAM is shown (in log scale).

5.4 B+Tree Benchmarks

We built a B+Tree data structure via Boost framework [1] using the in-built Boost *object_pool* allocator (which uses `malloc` internally). We then ported it to SSDAlloc OPP (in 15 lines of code) by replacing calls to `object_pool` with `ssd_oalloc`. We also ported it to MP by replacing all calls to `malloc` (inside `object_pool`) with `ssd_malloc` (in 6 lines of code). Hence, in the MP version, every access to memory happens via the SSDAlloc's runtime mechanism.

We use the Intel X25-V SSD (40GB) for the experiments and restrict the amount of memory in the system to 256MB for both the systems to test out-of-DRAM behavior. We allow up to 25 keys stored per inner node and 25 values stored in the leaf node, and we vary the key size. We first populate the B+Tree such that it has 200 million keys, to make sure that the height of the B+Tree is at least 5. We vary the size of the key, so that the size of the inner object and leaf node object vary. We perform 2 million *updates* (values are updated) and *lookups*.

Figure 5(b) shows that MP and OPP provide much higher performance than using SSD-swap. As the key size increases from 4 to 64 bytes, the size of the nodes increases from 216 bytes to 1812 bytes. The performance of SSD-swap and MP is constant in all cases (with MP performing **3.8** times better than SSD-swap with log-structured writes) because they access a full page for almost every node access, regardless of node size, increasing the size of the total dirty data, thereby performing more erasures on the SSD. OPP, in comparison, makes smaller reads when the node size is small and its performance scales with the key size in the B+Tree. We also report that across SSDs, B+Tree operations via OPP were **1.4–3.2** times faster when compared to MP and **4.3–12.7** times faster than when compared to SSD-swap (for a 64 byte key). In the next evaluation setting, we demonstrate how OPP makes the best use of DRAM transparently.

5.5 HashCache Benchmarks

Our final application benchmark is the efficient Web cache/WAN accelerator index based on HashCache [9]. HashCache is an efficient hash table representation that is devoid of pointers; it is a set-associative cache index with an array of sets, each containing the membership information of a certain (usually 8–16) number of elements currently residing in the cache. We wish to use an SSD backed index for performing HTTP caching and WAN Acceleration for developing regions. SSD backed indexes for WAN accelerators and data deduplicators are interesting because only flash can provide the necessary capacity and performance to store indexes for large workloads. A netbook with multiple external USB hard drives (upto a terabyte) can act as a caching server [8]. The inbuilt DRAM of 1–2 GB would not be enough to index a terabyte hard drive in memory, hence, we propose using SSDAlloc in those settings – the internal SSD can be used as a RAM supplement which can provide the necessary index lookup bandwidth needed for WAN Accelerators [16] which make many index lookups per HTTP object.

We create an SSD based HashCache index for 3 billion entries using 32GB SSD space. For creating the index, HashCache creates a large contiguous array of 128 byte sets. Each set can hold information for sixteen elements – hashes for testing membership, LRU usage information for cache maintenance and a four byte location of the cached object. We test three configurations of HashCache: with OPP (via `ssd_oalloc`), MP (via `ssd_malloc`) and SSD-swap (via `malloc`) to create the sets. In total, we had to modify 28 lines of code for these modifications. While using OPP we made use of *Checkpointing*. This is because we want to be able to quickly reboot the cache in case of power outages (netbooks have batteries and a graceful shutdown is possible in case of power outages).

Figure 6(a) shows, in log scale, the maximum number of useful index entries of a web workload (highly random) that can reside in RAM for each allocation method. With available DRAM varying from 2GB to 4.5GB, we show how OPP uses DRAM more efficiently than MP and SSD-swap. Even though OPP's Object Table uses almost 1GB more DRAM than MP's Object Table, OPP still is able to hold much larger working set of index entries. This is because OPP caches at set granularity while MP caches at a page granularity, and HashCache has almost no locality. Being able to hold the entire working set in memory is very important for the performance of a cache, since it not only saves write traffic but also improves the index response time.

We now present some reboot and recovery time measurements. Rebooting the version of HashCache built with OPP Checkpointing for a 32GB index (1.1GB Ob-

ject Table) took **17.66 sec** for the Kingston SSD (which has a sequential read speed of 70 MBPS).

We also report performance improvements from using OPP over MP and SSD-swap across SSDs. For SSDs with parallelism, we partition the index horizontally across multiple threads. The main observation is that using MP or SSD-swap would not only reduce performance but also undermine reliability by writing more number of times and more data to the SSD. OPP's performance is **5.3–17.1** times higher than when using SSD-Swap, and **1.3-3.3** times higher than when using MP across SSDs (50% insert rate).

6 Conclusion

SSDAlloc provides a hybrid memory management system that allows new and existing applications to easily use SSDs to extend the RAM in a system, while performing up to 17 times better than SSD-swap, up to 3.5 times better than log-structured SSD-swap and increasing the SSD's lifetime by a factor of up to 30 times with minimal code changes, limited to the memory allocation part of the application code. The performance of SSDAlloc based applications is close to that of custom-developed SSD applications. We demonstrate the benefits of SSDAlloc in a variety of contexts – a data center application (memcached), a B+Tree index, a packet cache backend and an efficient hashtable representation (HashCache), which required only minimal code changes, little application knowledge, and no expertise with the inner workings of SSDs.

7 Acknowledgments

We would like to thank our shepherd, Eddie Kohler, as well as the anonymous NSDI reviewers. This research was partially supported by the NSF Awards CNS-0615237, CNS-0916204 and CNS-0519829.

References

[1] Boost, . `http://www.boost.org/`.

[2] Calibrator, . `http://homepages.cwi.nl/~manegold/Calibrator/#6`.

[3] Scaling Memcaced at Facebook, . `http://www.facebook.com/note.php?note_id=39391378919`.

[4] Memcached, . `http://www.danga.com/memcached/`.

[5] ptmalloc, . `http://www.malloc.de/en/`.

[6] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, Apr. 2010.

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[8] A. Badam and V. S. Pai. Beating Netbooks into Servers: Making Some Computers More Equal Than Others. In *Proc. 3rd ACM Workshop on Networked Systems for Developing Regions (NSDR)*, BigSky, MO, 2009.

[9] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. Hashcache: Cache storage for the next billion. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.

[10] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. ASPLOS'92*, 1992.

[11] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *Operating Systems Review*, 42(2): 88–93, 2007.

[12] M. Castro, A. Adya, B. Liskov, and A. C. Myers. Hac: Hybrid adaptive caching for distributed storage systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, Oct. 1997.

[13] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[14] B. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[15] P. V. der Linder. *Expert C Programming: Deep C Secrets*. Prentice Hall, Englewood Cliffs, N.J, 1994.

[16] S. Ihm, K. Park, and V. S. Pai. Wide-area Network Acceleration for the Developing World. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[17] T. Kgil and T. N. Mudge. Flashcache: A NAND flash memory file cache for low power web servers. In *Proc. of CASES'06*, 2006.

[18] S. Ko, S. Jun, Y. Ryu, O. Kwon, and K. Koh. A New Linux Swap System for Flash Memory Storage Devices. In *In ICCSA'09*, 2008.

[19] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008.

[20] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for NVM+DRAM hybrind main memory. In *Proc. HotOS XII*, Monte Verita, Switzerland, May 2009.

[21] D. Narayanan, E. Thereska, A. Donelly, S. Elnikety, and A. Rowstron. Migrating server storage to ssds, analysis of tradeoffs. In *Proceedings of EuroSys'09*, 2009.

[22] 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.

[23] M. Saxena and M. M. Swift. Flashvm: Virtual memory management on flash. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[24] C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient b-tree layer for flash-memory storage systems. In *Proccedings of RTCSA'04*, 2004.

[25] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 1994.

Model Checking a Networked System Without the Network

Rachid Guerraoui and Maysam Yabandeh
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

Current approaches to model checking distributed systems reduce the problem to that of model checking centralized systems: *global* states involving all nodes and communication links are systematically explored. The frequent changes in the network element of the global states lead however to a rapid state explosion and make it impossible to model check any non-trivial distributed system. We explore in this paper an alternative: a *local* approach where the network is ignored, a priori: only the local nodes' states are explored and in a separate manner. The set of valid system states is a subset of all combinations of the node local states and checking validity of such a combination is only performed a posteriori, in case of a possible bug. This approach drastically reduces the number of transitions executed by the model checker. It takes for example the classic *global* approach several minutes to explore the interleaving of messages in the celebrated Paxos distributed protocol even considering only three nodes and a single proposal. Our *local* approach explores the entire system state in a few seconds. Our local approach does clearly not eliminate the state exponential explosion problem. Yet, it postpones its manifestations till some deeper levels. This is already good enough for online testing tools that restart the model checker periodically from the current live state of a running system. We show for instance how this approach enables us to find two bugs in variants of Paxos.

1 Introduction

At each step of model checking a centralized system, (i) one of the traversed states is selected, (ii) an enabled event is executed on that state, and (iii) the resulting state is added to the list of traversed states. The user-specified invariants are checked against the traversed states after each step and the set of these states grows exponentially with the *depth* of the exploration, i.e., the length of the sequence of enabled events considered. Current approaches to model checking distributed systems [7, 8, 18, 19, 14] reduce the problem to that of model checking a centralized system (Figure 1). The sets explored are *global* states comprising the *local* states of the nodes involved in the distributed system, i.e., the *system* state, as well as the *network* state involving the exchange of messages.

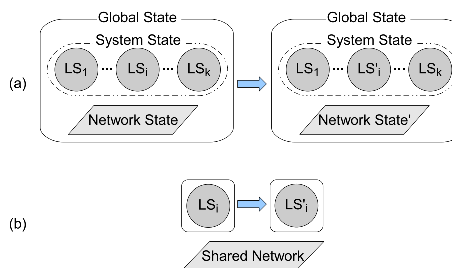


Figure 1: State transition in model checking distributed systems. In (a) the classic global approach, the model checker creates the entire state space of the global states, whereas in (b) our proposed local approach, the network element is eliminated from the stored states and the model checker keeps track of only node local states.

The exponential state space explosion problem manifests itself very quickly in this *global* approach, which makes the model checking of distributed systems practically ineffective. This is because the global state changes following any small change into a node local state or the network state. Consider for instance the celebrated Paxos protocol [9], in the simple setting with three nodes where exactly one proposes at the start, i.e., no contention: it takes the global model checking approach 1514 s (running on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache) to explore the interleaving of messages.

The starting point of this paper is a couple of simple, complementary observations: (1) in the global model checking approach, the invariants are checked on each

traversed global state, although these invariants are typically specified only on the system states, i.e., the invariants do not involve the network states [8, 18, 19, 14]; [1] (2) for checking invariants that are defined on system variables, visiting the system part is a priori sufficient. Focusing on these states only, and ignoring the network states, significantly reduces the exploration space in comparison to the classic approach where each system state is typically repeated in multiple global states that differ only in the network part.

We present in this paper a *local* model checking approach, which essentially consists in keeping track of the traversed local nodes' states separately by ignoring the network, a priori. Combined, these states are sufficient for invariant checking. The approach is most effective on protocols that involve frequent changes into the network, i.e., the nodes have lots of parallel network activities. For the Paxos example state space with one proposal, our approach explores the entire system state in a few seconds. We show that our approach is *complete* in the sense that any violation of a system state invariant that could be detected by the global approach could be detected by our local approach. Two important remarks are however in order.

First, the combination of node states does not induce system states that are all *valid*: the fact that we ignore the network element, a priori, means that some combinations of node states might not occur in a real run. In other words, although complete, checking invariants on the retrieved system states is *unsound* since it could report a violation on an invalid system state. We address this problem by, a posteriori, verifying every preliminary violation report to make sure the sequence of events leading to the corresponding system state could also happen in a real run. An invariant violation is then reported to the user only if passes this test. If the number of preliminary violations is low enough, which turns out to be the case in our experiments with Paxos, the performance penalty of verifying them becomes negligible.

Second, although our local approach is several orders of magnitude faster than the classic model checking approach, the state explosion problem is not eliminated. (The cost of invalid states created by our approach, although low at the start, will anyway eventually dominate in the general case.) Yet, we believe this can, to a large extent, be addressed by *online* model checking tools where the model checker is run for just a short period (a few seconds): in this case, our approach is efficient enough to search till depths of 20∼30 for the Paxos example state space.

[1] In testing, invariants are used to express the high-level properties of the system. Including the in-flight messages in invariants, although possible in theory, makes defining the invariants too complicated in practice.

In global model checking approach, visiting the system states is part of the exploration process: the new global states (which involve the system states) are explored by running enabled events on the previously visited global states. Therefore, skipping a system state makes the exploration incomplete. In contrast, our local approach separates the exploration of transitions from the creation of system states. This makes it possible to ignore all system states on which the user-specified invariants can inherently not be violated: for instance, the Paxos invariant stipulates that no two decisions should be different and all undecided states can systematically be eliminated.

Summary of Contributions.

- We introduce a new, local approach to model checking distributed systems. Instead of keeping track of global states, we eliminate the network element from the model checking states and keep only track of node local states. Our approach optimistically eliminates the overhead of ensuring soundness of every visited state and instead verifies soundness only on the states that violate the invariants.
- Our approach decouples exploration algorithm from system state space creation. This feature opens the door for optimizations that skip some system states without, however, hurting the completeness of exploration. We benefit from this aspect in our experiments by skipping the system states that could not violate the Paxos invariant.
- Having the exploration, system state creation, and soundness verification decoupled, the model checking process can be embarrassingly parallelized to benefit from the ever increasing number of cores.
- We present an efficient implementation of our approach and we show how this approach tracks bugs in two variants of Paxos, known to be one of the most complex distributed algorithms.

The rest of the paper is organized as follows. § 2 illustrates our approach through a simple example. The background is recalled in § 3. § 4 presents our approach. After presenting the evaluation results in § 5, we contrast local model checking approach with related work in § 6 and conclude the paper in § 7.

2 Local Model Checking: A Primer

Here we use a simple example to highlight the difference between global model checking and our local approach. The example we consider here does not attempt to illustrate the performance improvements obtained by our approach but aims at explaining the main idea. The example system is a simple distributed tree structure, depicted in Figure 2. Node 0 initiates a message for Node 4 and

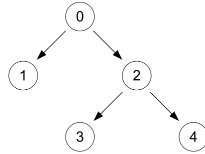


Figure 2: A simple distributed tree algorithm. Each node forwards the message to its children.

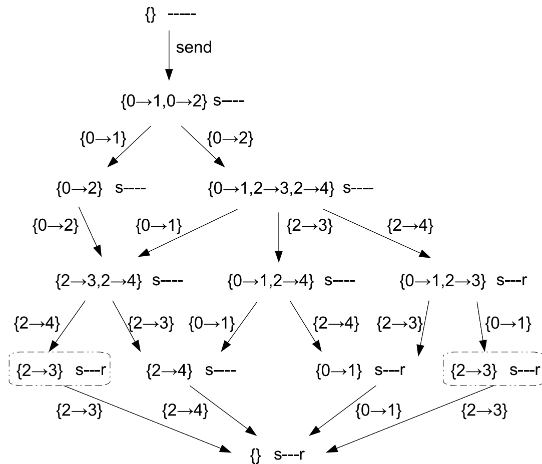


Figure 3: The global state space of the example tree in Figure 2 as explored by a global model checking approach. The network element of the global state is represented by the set of in-flight messages. Each arrow depicts a transition in the model checker from one global state to another. The label besides each arrow indicates the event that triggers the transition. Although the global states inside the rectangles are duplicates, they are not joined into one state, for simplicity of presentation.

changes its state to `sent`. Each node, upon receiving a message, forwards it to its children. Node 4 changes its state to `received` upon receiving the message.

At each step of global model checking, the model checker transitions from a global state to another by running an enabled event, such as handling a message. The global state contains the network state besides the system state, i.e., the local state of all the nodes. The global state space of the example system is depicted in Figure 3. The initial state of each node is denoted "-". The system state is shown by concatenating the five states of Nodes 0 to 4. The state of Node 0 and 4 is changed to "s" and "r" after the sending and receiving of the message, respectively. Each change into the network element causes creation of a new global state. As one can observe, the number of system states covered by this global state space is much less than its size.

Figure 4 illustrates our local approach on the same example system. Here, the network element, i.e., the non-

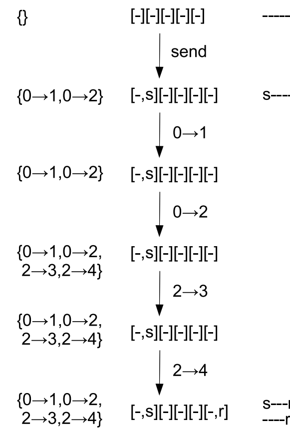


Figure 4: Local model checking approach on the example tree in Figure 2. The first column indicates the changes into the shared network element. The middle column shows the set of states of Node 0 to 4. The first event is the local event of Node 0 that generates the message. The generated message is then added to the shared network element. At each step, an event is selected and is executed on all states of the destination node. The resultant states are added to the list of visited node states if they have not been visited before. The last column shows the new system states created after each step.

essential part for invariant checking, is separated from the model checking state. Instead, we keep a shared network component that receives the generated messages by all the transitions in the model checking. Observe that the messages added to the network are not removed by the executed transitions. This is necessary for the completeness of the search, because each message must be received by all the states of the destination node, including the node states that will be explored later.

The last column of the figure depicts the new system states created after each step. The system states are created temporarily for the sake of being checked against the user-specified invariants. Observe that, in total, only 4 system states are created in contrast with the 12 global states of Figure 3. Moreover, the last system state, i.e., "----r" is invalid since Node 4 could not receive the message before it is sent by Node 0. After an invariant is violated on a system state, we run a *soundness* verification phase to ensure the validity of the system state.

3 Preliminaries

We present here a simple model of a distributed system and a basic model checking algorithm based on depth-first search. The model is later altered in § 4 to explain local model checking algorithm. We then explain the short

basic notions:
N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – internal node actions (timers, application calls)

global state : $(L, I) \in G, \ \ G = 2^{N \times S} \times 2^{N \times M}$
system state (local nodes' states) : $L \subseteq N \times S$
(function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \uplus c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I \uplus c)\end{array}}$$

Figure 5: A simple distributed system model

run in online model checking, where the model checker can benefit from our local model checking approach.

3.1 System Model

Figure 5 describes a simple model of a distributed system, taken from [18].

System state. The *global state* of the entire distributed system encompasses (1) the system state, i.e., local states of all nodes, and (2) in-flight network messages. We assume a finite set of node identifiers N (e.g., corresponding to IP addresses). Each node $n \in N$ has a local state $L^n \in S$. A node state encompasses node-local information, such as explicit state variables of the distributed node implementation, the status of timers, and the state that determines application calls. A network state corresponds to the set of in-flight messages, I. We represent each in-flight message by a pair (N, M) where N is the destination node of the message and M is the remaining message content (including sender node information and message body).

Node behavior. Each node in the system runs the same state-machine implementation. The state machine has two kinds of handlers: (i) a message handler executes in response to a network message; (ii) an internal handler executes in response to a node-local event such as a timer and an application call. We represent message handlers by a set of tuples H_M. The condition $((s_1, m), (s_2, c)) \in H_M$ means that, if a node is in state s_1 and it receives a message m, then it transitions into state s_2 and sends the set c of messages. Each element $(n', m') \in c$ is a message with target destination node n' and content m'. $((s_1, a), (s_2, c)) \in H_A$ represents the handling of an internal node action $a \in A$. An internal node action handler is analogous to a message handler, but it does not consume a network message.

System behavior. The behavior of the system specifies one step of a transition from one *global state* (L, I) to another global state (L', I'). We denote this transition by $(L, I) \rightsquigarrow (L', I')$ and describe it in Figure 5 in terms of handlers H_M and H_A. [2] The handler that sends the message, inserts the message directly into the network state I, whereas the handler receiving the message simply removes it from I. To keep the model simple, we assume that transport errors are particular messages, generated and processed by message handlers.

Observations. The following observations can be derived from the definitions of H_M and H_A in Figure 5: (i) Except the node in which the event is executed, the state of other nodes, i.e., L_0, is untouched. This implies that to execute an event on node n, we require only the state of node n; (ii) To execute H_M with message m on node n, the only required part from the network state is tuple (n, m): the rest of the network state, i.e., I_0, is untouched. These observations indicate that the entire global state of the system is not required to execute a handler in the model checker.

3.2 Global Model Checking

Global model checking is based on a standard search algorithm such as bounded depth-first search (B-DFS) for tracking invariant violations in the transition system captured by relation $\rightsquigarrow$ of Figure 5. The search starts from a given global state, which, in the standard approach, is the initial state of the system. By executing enabled handlers (H_M and H_A) on the traversed global states, the search systematically explores reachable global states at larger and larger depths and checks whether the states satisfy the given `invariant` condition.

Soundness. B-DFS is sound in the sense that all violation reports could also occur in a real run of the system. In other words, there is no *false positive* in the reported bugs. Moreover, all traversed states are valid and could also be created in a real run. The sufficient part for soundness, however, is only the reported violations

[2] $\uplus$ in the handler definition means disjoint union.

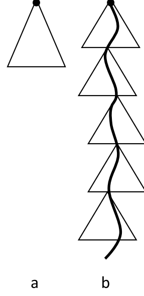


Figure 6: The covered state space in model checking by (a) a model checker started from the initial global state, and (b) an online model checker that restarts periodically from the current live system state. The curved line represents the states explored by the running system.

to the developer. We will show later that our local model checking is also sound, even though some system states created a priori might be invalid.

Completeness. An exploration algorithm is complete if, given enough time and space, it can explore all system states. In other words, completeness is satisfied if there is no *false negative* in bug reporting. Although B-DFS is complete, due to an inherently limited time budget, in practice it can explore only a small fraction of the state space of complex algorithms.

3.3 Online Model Checking

Due to the state space explosion problem, a model checker of a distributed system cannot explore deeper than certain steps in a limited time budget. For example, even in the very small state space experiment of Figure 10, where only one node proposes once, the model checker cannot explore more than 15 events within a minute. An online model checker is, on the other hand, restarted periodically from the live state of a running system. As a consequence, the model checker has a chance to explore more relevant states at deeper levels, instead of getting stuck in the exponential explosion problem at some very shallow depths.

Figure 6 illustrates the use of a model checker in parallel with a running system. As one can see, an online model checker does not require solving the exponential explosion problem completely; it is rather sufficient to explore till a depth that is useful for testing purposes.

4 Local Model Checking

The architecture of our local model checking approach is depicted in Figure 7. In this approach, the model checker keeps track of node states separately: set LS^n contains all the traversed states of node n. This is enough to

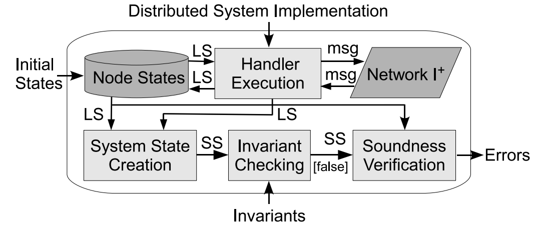


Figure 7: In our local approach, the handler execution works only on node states and produces new node states. Local and system states are denoted ”LS” and ”SS”, respectively. The messages are not removed from the shared network component after execution. The soundness verification checks the validity of a system state, only after an invariant violation is reported.

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H'_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I^+ \uplus \{(n, m)\}) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I^+ \uplus \{(n, m)\} \uplus c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H'_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I^+) \rightsquigarrow \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I^+ \uplus c)\end{array}}$$

Figure 8: The altered handlers in local model checking.

recreate the *system states* upon which the invariants are checked. After a preliminary violation report on a system state, the validity of the system state is checked by a soundness verification module. If the system state is confirmed to be valid, the error is then (and only then) reported to the developer.

Instead of keeping a separate network state for each global state, we keep one single network state I^+ that contains all generated messages during the model checking (Figure 7). The execution of handlers must change to work with the shared network state I^+ (Figure 8). In the new handlers, H'_M and H'_A, the network state of the input global state is replaced with the new shared network state, I^+. Furthermore, the received message, (n, m), is not removed from I^+ after the execution of handler H'_M. In other words, the content of I^+ is always increasing. It is not hard to see that the altered handlers preserve the completeness of the search: for each Transition $(L_p, I_p) \rightsquigarrow (L_q, I_q)$ in H_M, there exist a corresponding Transition $(L_p, I^+_p) \rightsquigarrow (L_q, I^+_q)$ in H'_M. We discuss soundness later in this section.

Recall from § 3 that, to execute a handler on node n, the only required state is the state of node n, i.e., LS^n.

```
1  proc findBugs(liveState, invariant)
2    LS = emptySet(); I+ = emptySet();
3    foreach n ∈ N
4      LS^n = LS^n ∪ {liveState^n};
5    while ( ! StopCriterion )
6      if ( ∃((s,e),(s',c)) ∈ H'_M where LS^n_s ∈ LS^n, (n,e) ∈ I+
7          || ∃((s,e),(s',c)) ∈ H'_A where LS^n_s ∈ LS^n)
8        addNextState(n, s, s', e, c, LS);
9        checkSystemInvariant(n, s', liveState, LS, invariant);
10
11 proc addNextState(n, s, s', e, c, LS)
12   I+ = I+ ∪ c;
13   LS^n = LS^n ∪ s';
14   LS^n_s'.predecessors.add(s, e);
15
16 proc checkSystemInvariant(n, s', liveState, LS, invariant)
17   foreach ss : system state
18       where ∀n_k. ss^{n_k} ∈ LS^{n_k}
19     if ( ! invariant(ss) )
20       if ( isStateSound(liveState, ss) )
21         reportBug(ss); // a bug found
22
23 proc isStateSound(liveState, state)
24   //obtain all sequences following predecessor pointers
25   foreach h : list of event sequences where
26     h^n ∈ (state^n.predecessors)* // * is closure operator
27     if ( isSequenceValid(liveState, h) )
28       return true;
29   return false;
30
31 proc isSequenceValid(liveState, h)
32   state = liveState;
33   while (∃n, nextState where state ~h^n.first()~> nextState)
34     state = nextState;
35     h^n.popFirst();
36   return h == ∅;
```

Figure 9: Local model checker algorithm.

Therefore, the stored node states are enough to execute the handlers and we do not need to recreate the system state for that. To execute network handlers, however, we require also message (n, m) from the network (we do not need the whole network state.). As shown in Figure 7, the handler execution module receives input only from node states and the shared network module.

4.1 Algorithm

Figure 9 presents our algorithm. Variable LS in Figure 9 refers to the set of all visited node states, i.e., (n,s), where n is the node index and s is the node state. Procedure `findBugs` takes the live state of the system as input, to initialize Variable LS at Lines 3-4. As in global model checking , the search terminates upon exceeding some bounds, such as running time or search depth (Line 5).

Handler execution. At each step of the model checking, an enabled handler, either network or local, is executed. For network handlers, the algorithm at each step checks all network messages in Variable I^+. To obtain the enabled network events, for each message e of node n in I^+, all the currently visited states of node n are considered (Line 6). The corresponding network handler is then executed (Line 8) and Procedure `addNextState` is called on the resultant state, s', and the set of new network messages, c. Note that the messages that are added to network I^+ in this round of the loop (i.e., c in Figure 8) will be considered on the node states in the next round.

As in the global model checking approach, the node local events, such as timers and application calls, are defined based on the node local states. In other words, the value of node state LS^n_s determines which of the local events are enabled. To obtain the enabled local events, we look at all visited node states and retrieve their local events (Line 7).

In Procedure `addNextState`, the set of new network messages is added to the shared network, I^+ (Line 12). If the state of node n has changed, it is added to set LS (Line 13). Variable $predecessors$ keeps track of all the last immediate node states as well as the executed events on them that led to the current node state (Line 14). We need more than one pointer in Variable $predecessors$, since the same node state might be reached by executing different sequences of events.

Creating system states. The invariants are defined on system states. Since we do not store the system states, they must be temporarily created for the sake of invariant checking, which is performed by Procedure `checkSystemInvariant`. The procedure is called after each change to LS. Each system state ss is created by combining the node states of different nodes in LS. (We will explain in § 4.2 an optimization that prevents revisiting system states.)

The only purpose of system state creation is to verify the user-specified invariant in on them. Therefore, we can design invariant-specific system state creation to bypass the system states that could not possibly violate the invariant. In other words, if $in' \Rightarrow in$ and $in'(ss)$ is false, verifying $in(ss)$ is not necessary. In order for this to be useful, in' should be cheaply verifiable. One way to achieve that is to decompose in' into some locally verifiable properties. For example, the Paxos invariant specifies that no two nodes should choose different values. In system state creation, therefore, we can ignore the node states in which no value is chosen yet. If the invariant is defined on node states separately, the invariant-specific system state creation can also bypass the system states in which none of node states have violated the invariant. For example, in RandTree distributed tree structure, one invariant specifies that in all node states the children and

siblings must be disjoint sets.

Soundness verification. Since taking all combinations of node states could result into some invalid system states, the preliminary violation of an invariant could be unsound. Procedure `isStateSound`, therefore, verifies validity of the system state upon which an invariant is violated. Variable $predecessors$ in each node state s' contains all the last immediate node states that led to s'. Following these pointers, we obtain the set of event sequences that could lead to s'. If a system state is valid, then there exists at least one valid combination of its node states' event sequences.[3] Lines 25-26 loop on all these combinations and invoke Procedure `isSequenceValid` on each. The number of paths could exponentially increase with sequence size, which is the major cost in soundness verification.

Procedure `isSequenceValid` receives n event sequences (h^i, $i \in N$) corresponding to n nodes in the system. The procedure then looks for a valid total order for execution of the events, in which an event is executed only after it is enabled. For example, to execute a network handler that receives message m from node s, the message must first be generated by an event in s. At each step, the procedure verifies whether any of the events on top of the h^i stacks are enabled (Line 33). The first enabled event is greedily selected for execution based on the definition of handlers in Figure 5 (the events are executed similar to a real run of the distributed system.). The loop continues until there are no enabled events on top the h^i stacks. Afterward, the fact that h is empty (Line 36) indicates that the set of sequenced events in h was possible to run and hence its corresponding system state is valid.

Procedure `isSequenceValid` returns true if and only if the corresponding input system state is valid. The proof of the above statement is covered in the technical report [16]. Intuitively, since an event in not popped out from h unless it is a valid, enabled event, the feasibility of executing all events implies that the system state is valid. It actually does not matter which enabled event is selected for the next step, since the demanded order by the sequences will be eventually enforced by receiving only the messages that are already generated.

4.2 Implementation Details

Local model checking can be used for testing programs in all languages, including C++. Basically, any of existing stateful global model checking tools could be instrumented to run our proposed algorithm. Our prototype implementation of the local model checking approach, denoted LMC, uses MaceMC [8], a model checker for distributed system implementations in the Mace language [7]. Mace programs are basically structured C++ implementations, in which the boundary of handlers and the protocol messages need to be specified. This helps Mace automatically generate the code for serialization and deserialization of the protocol state, and simplifies the definition of events in the model checker.

We use CrystalBall [18, 17] for online running of the model checker, in parallel with a live distributed system. The model checker is then periodically restarted from the taken snapshot. It is worth noting that LMC improves the performance of model checking anyway, independent of CrystalBall. For testing of complex programs, however, we use the online model checking approach to restart the model checker before exponential explosion manifests.

We changed MaceMC to work only on one global object of the network simulator, i.e., I^+. To change the network handler implementations from H_M to H'_M (Figure 8), we changed the network simulator not to remove a message after its delivery. MaceMC automatically generates specific functions for (de)serializing a module state in the service. We added specific functions to save and restore the whole service stack. This is required for multi-layer services such as 1Paxos [15] (one of the protocols we check), which uses Paxos as its lower layer module. To efficiently check for duplicate states, we use the hashes of the serialized states. For each node n, the hashes of the traversed states are kept in a `set` structure. The serialized state itself is stored in a `deque` structure to benefit from its efficiency in random access.

Each message keeps track of the number of node states on which it has been executed. Therefore, in each round, each message is checked only on the newly added states, by jumping over the old states. Instead of the actual event, its hash is added into the predecessor pointers. These hash values will be checked against the hash values of the enabled events, later when we verify the soundness of the system state.

Test driver. The test in model checking a service is generally driven by an application sending requests to the service. In Paxos for example, an application sending propose requests to the service is the test driver of the model checker. The more complex the test driver, the larger the generated state space is. A careful design of the test driver could greatly impact the efficiency of model checking. In our Paxos experiments, the test driver proposes values for a particular index. The index is selected from recent chosen proposals, where not all the nodes have learned the proposal yet. Otherwise, a new index is used for the proposal.

System states. To avoid revisiting system states, checking invariants on system states is performed only after

[3] Each event sequence must deterministically lead to the same node state. If the event handler implementation is dependent on some non-deterministic values, those values must be recorded as part of the event, to be replayed deterministically on a re-execution of the event.

visiting a new node state, which implies the possibility for creating new system states. For each new node state (n,s), the system states are created by iterating over the states of all the nodes except node n and loading them. This is because the combinations of the previously visited states of node n and the node states of the other nodes have already been verified in previous rounds. It is worth noting that this optimization could make the model checking incomplete because the handler execution that has not produced a new node state could still change the pointers in $predecessors$, which means the possibility of a valid event sequence for a previously rejected system state. To address this issue we could cache the system states in which an invariant is violated and reverify them after the changes into LS that affect them.

Beside the general approach for system state creation, we also implemented an invariant-specific variation, denoted LMC-OPT, optimized for the Paxos main invariant. In this variation, we map the node states to the values that are chosen in them. Because most of the node states have not chosen any value, lots of them will not be included in this mapping. When creating system states, we thus select only the node states that at least two of them are mapped to different values. This optimization helps avoid the creation of lots of redundant system states and consequently omits their corresponding invariant checking and soundness verification steps.

Soundness verification. Procedure `isStateSound` uses pointers in Variable $predecessor$ to find event sequences that could lead to the input node states. For the sake of simplicity in implementation, we ignore the *self-references* in following the pointers in $predecessor$. Although in theory this could make the exploration incomplete, in practice the search in the limited time budget is incomplete anyway and benefiting from the simplicity is, hence, preferable. Moreover, after the soundness verification on a system state is finished, some more pointers could be added into $predecessor$ by the process of local model checking. Therefore, a complete exploration should invoke soundness verification after each change into a $predecessor$. However, an efficient implementation of that would be complex since it should check only for the newly added pointers. For the sake of simplicity in implementation, we invoke soundness verification only after a new node state is visited.

Procedure `isSequenceValid`. The validity of a set of sequenced events could in general be checked by executing them in a simulator (the same way the global model checking approach transitions from one global state to another). If no event from the sequences is enabled in the simulator, it indicates that sequence of events is not valid. Although using the simulator simplifies the implementation, initializing the simulator at each run of the soundness verification is expensive since it involves loading the test driver.

For efficient implementation of soundness verification module, we take advantage of the following observation. The role of the simulator in executing event e on node n is to (i) updates the state of node n, (ii) remove the message m from the network if e is a network event for delivery of message m, and (iii) add the set c of messages, resulting from the execution of e, to the network.

The consumed message by a network event is specified by its corresponding hash in the node event sequence, which was given as a part of the input to the procedure. The set of the generated messages by an event execution can also be remembered by keeping the hashes of the generated messages in $predecessor$. In this manner, the input to Procedure `isSequenceValid` is the set of sequenced events as well as the set of generated messages by each event. The execution of event e in Procedure `isSequenceValid` can then be simplified as follows:

1. A local event e is always enabled. A network event e is enabled if the hash of the required message is found in the set of generated message hashes, net.
2. If event e is enabled, then pop it out from the sequence. If event e is a network event, remove the hash of the corresponding message from set net.
3. After popping out event e, add its generated message hashes to set net.

The above implementation simplifies Procedure `isStateSound` to some integer comparison operations and therefore makes checking the validity of a set of sequenced events very efficient.

Local assertions. LMC checks for the system invariants defined on the system state. The source code could still be instrumented by some local assertions by which the developers have benefited in earlier stages of testing. The violation of the local assert statements in the process of local model checking could imply that either (i) the node state is invalid, perhaps because of delivering an unexpected message, or (ii) there is a bug in the system under test. Checking the latter case necessitates (i) creating all the system states by combining the node state with all states from other nodes, and (ii) checking the validity of those states by invoking soundness verification. This approach is very expensive since it involves lots of invocation of soundness verification.

In general we could ignore violation of a local assert since a protocol bug will eventually manifest itself by violating a system invariant. Alternatively, we can discard the node state on which the assertion is violated assuming that the assert violation implies the invalidity of the node state. In the applications we tested, the assert statements were mostly used to exclude the receipt of unexpected messages, i.e., the case that could be caused by conservative message delivery policy of LMC, which de-

livers the message to all the node states of the destination. We, therefore, benefited from the local assert violations by discarding the corresponding node states.

Local events. The presented algorithm in § 4 is complete in the sense that, given enough time and space, it explores all possible states. In practice, however, we have a short time budget to check the reachable states from a given current state. Therefore, the developers might be interested to favor some events to be explored first in the search. Hence, in each round we put a bound on the number of local events that each node can execute; after finishing the round, the bounds are increased and the model checking is started from scratch. This approach is in spirit similar to B-DFS search, where the search depth is increased at each step.

Duplicate messages. In general, a node could infinitely issue duplicates of the same message. For example, in the verified Paxos implementations, the same Chosen message will be sent over and over to the proposer that insists for an already chosen value. To favor the main protocol messages in the limited time of search, we have put a limit on the number of duplicate messages sent from a source to a destination node. This limit is set to zero for the results reported in this paper. Note that the duplicate messages can be postponed to be processed later, after processing some main protocol messages.

As we explained, to ensure completeness, the messages are never erased from the network object, I^+. However, if node state $s \overset{m}{\rightsquigarrow} s'$ where m is a network event, execution of m on s' is redundant since m is already executed in the sequence. To avoid redundant executions, we keep the history of the messages that has been executed to obtain the state: a network event is considered on a state only if it is not in the history of the state. After executing message m on node state s that results into node state s', we apply the two following rules to maintain the history: (i) $s'.history = s.history$, (ii) $s'.history.addLast(m)$. Thus, message m will never be executed on node state s' as well as its descendants. Maintaining history gets complicated if state s' already exists since we need to maintain separate histories for different sequences that lead to s'. We have simplified the implementation by applying rule (i) only if the state does not exist. Since the run of LMC in the limited time budget is not complete anyway, we decided to favor simplicity over completeness here.

4.3 Scope of Applicability

In contrast with global model checking that validity of each traversed state is ensured, local model checking optimistically allows visiting invalid states and verifies the validity of a state only after it violates an invariant. If we have a few preliminary violations, the optimistic approach of local model checking performs better since it does not pay for ensuring validity of every single visited state. Otherwise, the cost of soundness verification dominates. For example, in online model checking, if a run of the model checker is revealing a bug in the protocol, it is likely to see lots of violation reports caused by both valid and invalid event sequences. Perhaps, one solution could be running both local and global model checker in parallel and use the result of the one that finishes sooner.

By eliminating the network element from the model checking state, local model checking reduces the explored state space since each system state is repeated in multiple global states that are different only in the network part. The larger the network state space is, the more space and time is saved by eliminating it. Local model checking is, therefore, most effective for the protocols that are chatty, i.e., exchange lots of messages to service a request. Otherwise, if the nodes rarely communicate, the change into the network is rare and therefore there is not much to be saved by local model checking.

In contrast with global model checking, local model checking considers interleaving of parallel network events only when they turn out to be dependent. LMC, therefore, avoids lots of unnecessary event interleaving. For example, upon receipt of the Accept message, the nodes in Paxos broadcast some Learn messages in parallel, which enables LMC to perform much better than global model checking. The more parallel network activities in the system, the more effective LMC is. For example, we could not expect much from LMC in a chain system in which each node simply forwards the input message to the next.

The current implementation of LMC assumes a best-effort, lossy network, i.e., IP. The protocols that use UDP can, therefore, be directly model checked with LMC. Although, TCP could be considered as part of the protocol stack, in practice this is not efficient, and TCP is usually simulated in the model checker. To do so, LMC implementation should be also augmented to benefit from the fact that reordered messages in a connection will eventually be rejected by TCP and could, hence, be ignored, saving some unnecessary handler executions in the model checker.

5 Evaluation

We evaluate in this section the performance of our local model checking approach compared to a classic global one. We also illustrate the ability of our tool, LMC, in finding bugs in Paxos and its variant, 1Paxos.

We use Paxos as a complex distributed testbed to evaluate the performance of the proposed local model checking approach. In usual implementations of Paxos, each node implements three roles: proposer, acceptor, and

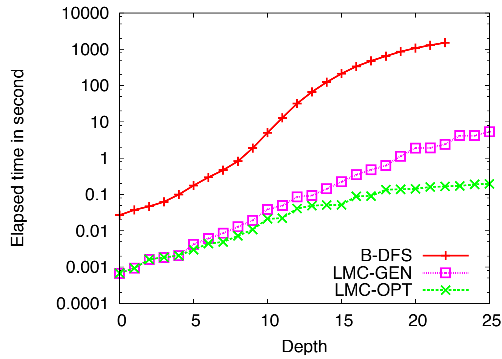


Figure 10: The elapsed time in model checking Paxos where only one out of three nodes proposes a value.

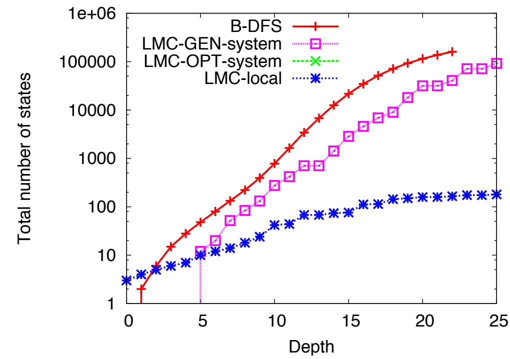


Figure 11: The number of explored states. The number of system states explored by LMC-OPT is zero and is, hence, not plotted in the figure.

learner. Multiple proposers can concurrently propose values for the same index. The Paxos invariant (also known as the Paxos safety property) stipulates that no two nodes will choose different values for the same index. A proposition (i.e., proposing a value for an index) starts by broadcasting Prepare messages to the acceptors. The acceptors respond by a PrepareResponse message. After receiving it from a majority of acceptors, the proposer broadcasts an Accept message to the acceptors. The value in the Accept message is the value returned by the PrepareResponse message with the highest proposal number, which reflects the accepted values from previous proposals, if there is any. Each acceptor then broadcasts a Learn message to the learners. A value is chosen by the learners after receiving the Learn message from a majority of acceptors.

For benchmarking purposes, we use a state space of Paxos running between three nodes, in which one node proposes a value once and the others react to this proposal by communicating using Paxos messages. The long chain of messages following each proposal could be received in a variety of orders, which all must be considered by a model checker. For each experiment, we report on evaluation of 3 algorithms: (i) B-DFS (explained in § 3), (ii) LMC-GEN, which is the non-optimized, general version of our local model checker (LMC), and (iii) LMC-OPT, which is a version of our local model checker optimized for the Paxos main invariant according to § 4.2. The experiments are run on a 3.00 GHz Intel(R) Pentium(R) 4 CPU with 1 MB of L2 cache.

5.1 LMC Speedup

Here we evaluate the speedup in model checking that we can get by our tool, LMC. Figure 10 presents the results for the example state space, in which only one node proposes a value. This state space is relatively small and yet effective in finding bugs when it is explored through an online model checker. The depth of the state space is 22 events (three initialization, one propose local event, three Prepare messages, three PrepareResponse messages, three Accept messages, and nine Learn messages). LMC explores also longer sequences of events (up to 25) since it could also explore some invalid sequences of events. [4] The elapsed time is depicted in a logarithmic scale to illustrate exponential state space explosion problem. In B-DFS, the exponential explosion starts from the very early steps, which makes the exploration take 1514 s. The growth in LMC-OPT is much less steep, which allows it to finish the model checking in just 189 ms (∼8,000 times faster than B-DFS).

The growth in LMC-GEN, although still much more gentle than B-DFS, is steeper than LMC-OPT. The exploration finishes in 5.16 s which is still ∼300 times faster than B-DFS. The extra delay is due to the creation of the system states out of the explored node states, which in LMC-OPT is optimized to be performed only after a different value is chosen. Figure 11 depicts the number of explored states. The number of created system states in LMC-GEN, although much less than B-DFS, is much more than the total number of node states, denoted LMC-local in the figure. LMC-OPT, on the other hand, drops the number of created system states to zero since there is no bug in the Paxos implementation to lead to any preliminary violations. (LMC-OPT creates a system state only if it is likely to invalidate the invariants.)

The total number of performed transitions in B-DFS is 157,332. LMC drops this to 1,186, which is ∼132 times less. This is because a LMC transition from state s to state s' in node n, is redundantly executed several times in global model checking approach (once for each global state that encompasses s and its network event is enabled).

[4] The invalid sequences will be eventually rejected by soundness verification phase if they violate some invariants.

This state space of Paxos is very useful in online model checking, where we expect the model checker to seek for a bug in the time budget of less than a minute. Both LMC-OPT and LMC-GEN can finish this state space in this duration and LMC-OPT can continue for more complicated state spaces where there is some time left (as we explained in § 4.2, the model checker, in favor of time, starts with small state spaces by gradually increasing the number of allowed local events.). This is in contrast to B-DFS that will not go further than depth 12 within a minute.

5.2 LMC Scalability Limits

We showed that LMC manages to finish a valuable state space in less than a few seconds. This is already good enough for practical applications such as online model checking that restarts the model checker every few seconds. From the theoretical point of view at least, it is interesting to find the scalability limits of LMC, i.e., the point where the postponed exponential explosion problem eventually manifests and makes LMC ineffective for the rest of the exploration. To this aim, we choose a much bigger state space, where two separate nodes propose two values. The depth of the state space is 41 events, which is two times the events in one error-free proposal. (LMC explores also longer sequences of events, up to 68, since it could also explore invalid sequences of events.)

Due to exponential explosion problem, neither B-DFS nor LMC could finish the state space, even after hours of running. Within this duration, B-DFS explores till 20 steps (out of maximum depth of 41) and LMC searches till 39 steps (out of maximum depth 68). The major contributor to the slowdown of LMC is the expensive task of soundness verification. The number of different event sequences that must be considered for checking validity of a system state exponentially increases with the search depth. In the above example that the search depth of LMC is 39, each invocation of soundness verification induces ∼10 s into the algorithm. Invocations of soundness verification are much less in the smaller state space in which only one node proposes a value.

5.3 LMC Memory Requirements

Figure 11 depicts the very fact that the number of node states explored by LMC is much less than the total number of system or global states. Because LMC keeps track only of node states, and the system states are created only temporarily, LMC is expected to require very low memory footprint. Figure 12 verifies this expectation by depicting the memory footprints of different algorithms. LMC-local denotes the run of LMC-OPT in which the creation of system states is disabled. The difference between LMC-local and LMC-OPT (resp. LMC-GEN) indicates the memory overhead of system state creation as well as soundness verification. Although there is a marginal overhead for system states, the memory eventually returns to the system by reusing the deleted objects. The consumed additional memory by all algorithms is less than 1 MB which can totally fit into the L2 cache. However, the exponential trend in memory consumption of B-DFS, promises the ineffectiveness of B-DFS for deeper searches. LMC in contrast uses the memory very efficiently (∼200 KB in total) and this amount grows linearly by increase in search depth.

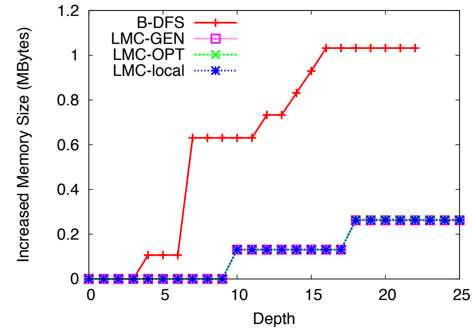


Figure 12: The consumed memory. The numbers for all configurations of LMC are close together and are, hence, overlapped in the figure.

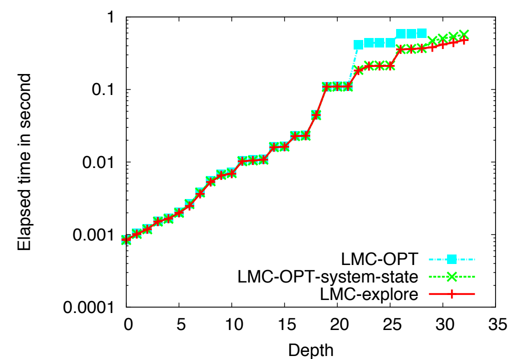


Figure 13: The overheads of LMC in model checking Paxos in which a bug is injected.

5.4 LMC Overheads

Here we break down the overheads that limit the scalability of LMC. LMC has two major overheads: (1) creation of system states out of traversed node states, and (2) verifying soundness of the preliminary violations. The precise load of each overhead depends on the particular system under test. Figure 13 illustrates the overheads of LMC-OPT in the buggy implementation of Paxos,

for which the corresponding bug is reported in § 5.5. In LMC-system-state the soundness verification phase is disabled and in LMC-explore the creation of system states is eliminated.

The difference between LMC-system-state and LMC-explore captures the overhead of creating the system states and checking the invariant on them. The overhead is zero until 21 steps since the unnecessary system states are bypassed by the optimization in LMC-OPT. Afterwards, the overhead increases with the depth search, because as the exploration moves forward, more node states are explored and hence more combinations of them must be considered for system state creation. The difference between LMC-OPT and LMC-system-state reveals the overhead of soundness verification. (LMC-OPT did not go further than 28 steps, the level at which the injected bug is rediscovered.) This overhead is the major contributor to the exponential increase in model checking time. The reason is that not all combinations of node states are valid, and the more node states are traversed, the more invalid system states will be checked. On the other hand, since the injected bug is close to manifest in this run of the model checker, the number of invalid combinations of node states that violate the invariant increases. LMC-OPT triggers the soundness verification for 773 times, and each call takes 45 ms in average. Overall, 427,731 different event sequences were checked by the soundness verification module.

5.5 Testing Paxos

In this section, we report on our experiments in injecting a bug into a Paxos implementation and then running our prototype to verify its ability to detect the bug. The bug we injected was reported in a previous implementation of Paxos [10]: once the leader receives the PrepareResponse message from a majority of nodes, it creates the Accept request by using the submitted value from the last PrepareResponse message instead of the PrepareResponse message with highest round number. The installed invariant is the original Paxos invariant: no two nodes can choose different values.

Every one minute, the online model checking framework takes the live system state of a running Paxos application and use that to initialize the next run of LMC. The application encompasses three nodes, each node proposes its Id for a new index and then sleeps for a random time between 0 and 60 s. The nodes communicate using UDP and 30% of non-loopback messages are randomly dropped to allow rare states to be also created.

The bug was detected after 1150 seconds. The run of LMC that detected the bug was initialized with the following live state: for index k_i, node N_1 has proposed value v_1, nodes N_1 and N_2 have accepted this proposal, but due to message losses only N_1 has learned it. Starting from this system state, LMC detected in 11 s a violation of the Paxos invariant in the following scenario: N_2 proposes a new value v_2 but its Prepare messages is not received by N_1. N_2 responds by a PrepareResponse message containing value v_1, because this value was accepted by N_2 in the previous round. However N_3, since had not accepted any value for index k_i, responds back by the same value proposed by N_2, v_2. Receipt of PrepareResponse of N_3 triggers the bug, and N_2 broadcasts an Accept message for v_2 instead of v_1. Eventually this leads to choosing value v_2 in N_2, which is different from the value chosen by N_1, i.e., v_1.

5.6 Testing 1Paxos

In this section, we report on running our prototype to find bugs on a variant of Paxos, denoted 1Paxos [15]: this is an efficient variation of Multi-Paxos [2] that uses only one acceptor. Upon failure, the active acceptor is replaced with a backup acceptor by the global leader. Therefore, it is necessary that the acceptor and leader roles to be assigned to two separate nodes. To uniquely identify the global leader and the active acceptor, 1Paxos uses a separate consensus protocol referred to as PaxosUtility [15]. The global leader and the active acceptor are identified by the last LeaderChange and AcceptorChange entries in the PaxosUtility, respectively. In this experiment, we have implemented PaxosUtility using Paxos itself. 1Paxos is more complex than Paxos for it comprises more logic. Here we use the same setup that was used for testing Paxos, with the difference that the application instead of proposing a value triggers the fault detector with the probability of 0.1 to stress the fault tolerance mechanisms of 1Paxos. In 225 s, the tool found one new bug in 1Paxos that we report in the following.

The bug was created because of the wrong usage of the "++" operator; if the operator is used after the operand, the returned value is the original value and not the increased one. The developer had made this mistake in the initialization function, where the leader is set to the first node of the members and the acceptor is set to the second. The used command was `acceptor = *(members.begin()++)` which makes the acceptor be the same node as the leader. The bug is of course fixed by putting the "++" operator before the operand, i.e., `acceptor = *(++members.begin())`.

During the live run, node N_3 attempts to be the leader by inserting a LeaderChange entry into the PaxosUtility. At this moment, it obtains from the PaxosUtility the correct value of the active acceptor, which is N_2. After N_3 becomes leader, it proposes value v_3 for index k_i, which is accepted by the acceptor, i.e., N_2. N_2 then broadcasts a Learn message, which is received by N_3 as well as it-

self. At this point the live system state, in which all nodes except N_1 have chosen value v_3 for the index k_i, is taken to be used by LMC.

Starting from the above system state, LMC highlights the following scenario that violates the Paxos invariant: N_1, which still assumes it is the leader, proposes value v_1 for index k_i to the acceptor. Since N_1 considers itself to be the leader, according to the protocol, it does not refer to PaxosUtility to get the acceptor Id. Therefore, N_1 uses its current value, which is set to N_1, i.e., its own Id, due to the initialization bug described above. N_1 accepts the proposal and sends a Learn message to N_1. Upon receiving the loopback message, N_1 assumes value v_1 as chosen for index k_i. This violates the Paxos invariant since other nodes have chosen a different value, i.e., v_3.

6 Related Work

Cartesian abstraction. This is an abstraction-based verification technique where an overapproximated variant of the program is model checked, instead of the original one [1]. Due to overapproximation, the reported bugs are not sound, which makes the technique mainly useful for correctness proving, benefiting from the completeness of the search. Malkis et al. [11] achieved thread-modular model checking [5, 12] using a Cartesian abstract interpretation of multi-threaded programs. Each thread state consists of the thread local variables plus the global variables. For each thread, the model checker separately explores possible valuations of the thread local variables as well as the global variables. The approximation comes from the fact that the valuations of the global variables by a thread are also used by other threads, ignoring the causal order for obtaining them. Again, the unsoundness, stemmed from the approximation, makes the technique inappropriate for testing purposes. In contrast, our reported bugs are sound and this is ensured by keeping track of the events executed for obtaining a node state and checking the validity of the combination of these histories after a preliminary invariant violation report.

We also make use of the Cartesian product of independently explored node states to obtain the system states. Cartesian abstraction is essential here in our approach in order to create the system states and check (system-wide) invariants against them. In contrast, previous works benefited from the Cartesian abstraction by not creating system states; skipping the system states is possible since the invariants in multi-threaded programs are just thread-local assert statements and could be verified on a local state of a thread without having the rest of the system state. [5] Our local model checking approach employs the Cartesian abstraction in a different way: namely, to explore the system state space without exploring the global state space.

In [6], Cartesian Abstraction is used on top of boolean abstraction of threads to find race conditions in multi-threaded programs. After boolean abstraction, each thread is represented by a long boolean expression over global and local variables including an artificially added variable for line number. A race condition is also represented by a boolean expression over the line numbers in which the threads read and write the global variables. Race conditions are detected by taking conjunction of the thread boolean expressions with race conditions. Therefore, there is no need for system state creation. This approach cannot be applied on general system invariants that would express a relation between local variables of multiple threads. The approach applies a heuristic on the detected races to eliminate some of the false positives.

One could indeed generalize the Cartesian abstract interpretation presented in [11] to distributed systems, by using the network as the global object. However, the network would still be part of the model checking states, concatenated to the local states. In our approach, we exclude the network element from the model checking state and use only a shared network element.

Monotonic abstraction. Monotonic abstraction [13] of the network has been used in verification of security protocols since it accounts for the maximal knowledge learned by attacker. Dolev-Yao's model [4] is one such model, in which the attacker remembers all messages that have been intercepted or overheard. The shared network object in our local model checking approach is essentially an application of a monotonic abstraction since the delivered messages are not removed from the network. The shared monotonic network is key to ensuring the completeness of the search by applying the generated messages also on future generated node states.

Online model checking. CrystalBall [18, 17] is a framework that implements the online model checking scheme. To be effective in practice, the online model checker must be fast enough to explore till a reasonable depth in the period between two restarts (typically a few seconds). CrystalBall uses a heuristic, namely *Consequence Prediction*, which prunes the local events of an already visited node state. As a heuristic, Consequence Prediction is incomplete and could, hence, miss some bugs due to false negatives. In contrast, our local model checking approach offers a complete search accompanied with proofs. Furthermore, complex distributed systems such as Paxos, often generate lots of network messages on which Consequence Prediction does not have any effect. For instance, in the used Paxos state spaces

[5] There is an ongoing research to convert a system-wide invariant to a set of thread-local assert statements, which has shown good results on small multi-threaded programs [3].

throughout this paper, we consider only the interleaving of the resulting network messages after some proposals. Therefore, Consequence Prediction, which does not prune the network messages, would not offer any improvement over B-DFS.

7 Concluding Remarks

We introduce a novel, *local* approach to model checking distributed systems. Essentially, the underlying idea is to remove the network state from the global state when model checking, and focus on the remaining system state, which is the usual required part for invariant checking. The system state is itself built temporarily out of node states, and these are maintained separately. Although complete, the approach is not sound in the sense that some system states could be invalid, i.e., could not have been produced by an actual run of the system. We check the soundness of the system state, a posteriori, only if an invariant is violated.

By removing the network from the global states, our local model checking approach creates much less system states than in the global approach. In addition, and in contrast with the latter approach, in which visiting the system states is an inherent part of the exploration process, local approach separates the exploration of transitions from the actual creation of system states. This makes it possible to exploit the specificities of the user-specified invariants and a priori eliminate all system states on which these invariants cannot be violated.

Clearly, the state exponential explosion problem is not eliminated in our approach, and it indeed eventually manifests, especially because of invalid system states. Yet the problem is postponed and this makes our local approach an adequate match for online model checking that restarts the model checker periodically. Using online model checking augmented with our local approach, we found a previously reported bug in a traditional Paxos implementation, as well as a new bug in a recent variant of Paxos. Both bugs have been identified by focusing on a simple, arguably common case, namely the case with no contention for which distributed protocols are typically optimized and hence error-prone.

For future works, one can think of methods to automatically prune the system states according to a given invariant. In addition, the low memory consumption of our approach brings potentials for techniques that trade memory for CPU, gaining more speedup.

8 Acknowledgments

We thank Viktor Kuncak for invaluable comments. We are also thankful to our shepherd Alex Snoeren and the anonymous reviewers for their excellent feedback.

References

[1] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, 2001.

[2] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[3] A. Cohen and K. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 2009.

[4] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Trans. on information theory*, 29(2), 1983.

[5] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking Software*. Springer, 2003.

[6] T. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *CAV*, 2003.

[7] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[8] C. E. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[9] L. Lamport. The part-time parliament. *TOCS*, 1998.

[10] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[11] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *ICTAC*. Springer, 2006.

[12] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-Modular Counterexample-Guided Abstraction Refinement. In *SAS*, 2010.

[13] J. Mitchell. Multiset rewriting and security protocol analysis. In *Rewriting Techniques and Applications*, 2002.

[14] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 2002.

[15] M. Yabandeh, L. Franco, and R. Guerraoui. One Acceptor is Enough. Technical report, EPFL, 2010.

[16] M. Yabandeh and R. Guerraoui. Local Model Checking. Technical report, EPFL, 2011.

[17] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. In *NSDI*, 2009.

[18] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. Predicting and preventing inconsistencies in deployed distributed systems. *ACM TOCS*, 28(1), 2010.

[19] J. Yang and et al. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, 2009.

FATE and DESTINI: A Framework for Cloud Recovery Testing

Haryadi S. Gunawi, Thanh Do[†], Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein,
Andrea C. Arpaci-Dusseau[†], Remzi H. Arpaci-Dusseau[†], Koushik Sen, and Dhruba Borthakur[*]

University of California, Berkeley [†] University of Wisconsin, Madison [*] Facebook

Abstract

As the cloud era begins and failures become commonplace, failure recovery becomes a critical factor in the availability, reliability and performance of cloud services. Unfortunately, recovery problems still take place, causing downtimes, data loss, and many other problems. We propose a new testing framework for cloud recovery: FATE *(Failure Testing Service) and* DESTINI *(Declarative Testing Specifications). With* FATE*, recovery is systematically tested in the face of multiple failures. With* DESTINI*, correct recovery is specified clearly, concisely, and precisely. We have integrated our framework to several cloud systems (e.g., HDFS [33]), explored over 40,000 failure scenarios, wrote 74 specifications, found 16 new bugs, and reproduced 51 old bugs.*

1 Introduction

Large-scale computing and data storage systems, including clusters within Google [9], Amazon EC2 [1], and elsewhere, are becoming a dominant platform for an increasing variety of applications and services. These "cloud" systems comprise thousands of commodity machines (to take advantage of economies of scale [9, 16]) and thus require sophisticated and often complex distributed software to mask the (perhaps increasingly) poor reliability of commodity PCs, disks, and memories [4, 9, 17, 18].

A critical factor in the availability, reliability, and performance of cloud services is thus how they react to failure. Unfortunately, failure recovery has proven to be challenging in these systems. For example, in 2009, a large telecommunications provider reported a serious data-loss incident [27], and a similar incident occurred within a popular social-networking site [29]. Bug repositories of open-source cloud software hint at similar recovery problems [2].

Practitioners continue to bemoan their inability to adequately address these recovery problems. For example, engineers at Google consider the current state of recovery testing to be behind the times [6], while others believe that large-scale recovery remains underspecified [4]. These deficiencies leave us with an important question: How can we test the correctness of cloud systems in how they deal with the wide variety of possible failure modes?

To address this question, we present two advancements in the current state-of-the-art of testing. First, we introduce FATE (Failure Testing Service). Unlike existing frameworks where multiple failures are only exercised randomly [6, 35, 38], FATE is designed to *systematically* push cloud systems into many possible failure scenarios. FATE achieves this by employing *failure IDs* as a new abstraction for exploring failures. Using failure IDs, FATE has exercised over 40,000 unique failure scenarios, and uncovers a new challenge: the exponential explosion of multiple failures. To the best of our knowledge, we are the first to address this in a more systematic way than random approaches. We do so by introducing novel prioritization strategies that explore non-similar failure scenarios first. This approach allows developers to explore distinct recovery behaviors an order of magnitude faster compared to a brute-force approach.

Second, we introduce DESTINI (Declarative Testing Specifications), which addresses the second half of the challenge in recovery testing: specification of expected behavior, to support proper testing of the recovery code that is exercised by FATE. With existing approaches, specifications are cumbersome and difficult to write, and thus present a barrier to usage in practice [15, 24, 25, 32, 39]. To address this, DESTINI employs a relational logic language that enables developers to write clear, concise, and precise recovery specifications; we have written 74 checks, each of which is typically about 5 lines of code. In addition, we present several design patterns to help developers specify recovery. For example, developers can easily capture facts and build expectations, write specifications from different views (*e.g.*, global, client, data servers) and thus catch bugs closer to the source, express different types of violations (*e.g.*, data-loss, availability), and incorporate different types of failures (*e.g.*, crashes, network partitions).

The rest of the paper is organized as follows. First, we dissect recovery problems in more detail (§2). Next, we define our concrete goals (§3), and present the design and implementation of FATE (§4) and DESTINI (§5). We then close with evaluations (§6) and conclusion (§7).

2 Extended Motivation: Recovery Problems

This section presents a study of recovery problems through three different lenses. First, we recap accounts of issues that cloud practitioners have shared in the literature (§2.1). Since these stories do not reflect details, we study bug/issue reports of modern open-source cloud systems (§2.2). Finally, to get more insights, we dissect a failure recovery protocol (§2.3). We close this section by reviewing the state-of-the-art of testing (§2.4).

2.1 Lens #1: Practitioners' Experiences

As well-known practitioners and academics have stated: "the future is a world of failures everywhere" [11]; "reliability has to come from the software" [9]; "recovery must be a first-class operation" [8]. These are but a glimpse of the urgent need for failure recovery as we enter the cloud era. Yet, practitioners still observe recovery problems in the field. The engineers of Google's Chubby system, for example, reported data loss on four occasions due to database recovery errors [5]. In another paper, they reported another imperfect recovery that brought down the whole system [6]. After they tested Chubby with random multiple failures, they found more problems. BigTable engineers also stated that cloud systems see all kinds of failures (*e.g.*, crashes, bad disks, network partitions, corruptions, etc.) [7]; other practitioners agree [6, 9]. They also emphasized that, as cloud services often depend on each other, a recovery problem in one service could permeate others, affecting overall availability and reliability [7]. To conclude, cloud systems face *frequent*, *multiple* and *diverse* failures [4, 6, 7, 9, 17]. Yet, recovery implementations are rarely tested with complex failures and are not rigorously specified [4, 6].

2.2 Lens #2: Study of Bug/Issue Reports

These anecdotes hint at the importance and complexity of failure handling, but offer few specifics on how to address the problem. Fortunately, many open-source cloud projects (*e.g.*, ZooKeeper [19], Cassandra [23], HDFS [33]) publicly share in great detail real issues encountered in the field. Therefore, we performed an in-depth study of HDFS bug/issue reports [2]. There are more than 1300 issues spanning 4 years of operation (April 2006 to July 2010). We scan all issues and study the ones that pertain to recovery problems due to hardware failures. In total, there are 91 recovery issues with severe implications such as data loss, unavailability, corruption, and reduced performance (a more detailed description can be found in our technical report [13]).

Based on this study, we made several observations. First, most of the internal protocols already anticipate failures. However, they do not cover all possible failures, and thus exhibit problems in practice. Second, the number of reported issues due to multiple failures is still small. In this regard, excluding our 5 submissions, the developers only had reported 3 issues, which mostly arose in live deployments rather than systematic testing. Finally, recovery issues appeared not only in the early years of the development but also recently, suggesting the lack of adoptable tools that can exercise failures automatically. Reports from other cloud systems such as Cassandra and ZooKeeper also raise similar issues.

2.3 Lens #3: Write Recovery Protocol

Given so many recovery issues, one might wonder what the inherent complexities are. To answer this, we dissect the anatomy of HDFS write recovery. As a background, HDFS provides two write interfaces: write and append. There is no overwrite. The write protocol essentially looks simple, but when different failures come into the picture, recovery complexity becomes evident. Figure 1 shows the write recovery protocol with three different failure scenarios. Throughout the paper, we will use HDFS terminology (*blocks*, *datanodes/nodes*, and *namenode*) [33] instead of GoogleFS terminology (chunks, chunk servers, and master) [10].

• **Data-Transfer Recovery:** Figure 1a shows a client contacting the namenode to get a list of datanodes to store three replicas of a block (`s0`). The client then initiates the setup stage by creating a pipeline (`s1`) and continues with the data transfer stage (`s2`). However, during the transfer stage, the third node crashes (`s2a`). What Figure 1a shows is the correct behavior of data-transfer recovery. That is, the client recreates the pipeline by excluding the dead node and continues transferring the bytes from the last good offset (`s2b`); a background replication monitor will regenerate the third replica.

• **Data-Transfer Recovery Bug:** Figure 1b shows a bug in the data-transfer recovery protocol; there is one specific code segment that performs a bad error handling of failed data transfer (`s2a`). This bug makes the client wrongly exclude the good node (Node2) and include the dead node (Node3) in the next pipeline creation (`s2b`). Since Node3 is dead, the client recreates the pipeline only with the first node (`s2c`). If the first node also crashes at this point (a multiple-failure scenario), no valid blocks are stored. This implementation bug reduces availability (*i.e.*, due to unmasked failures). We also found data-loss bugs in the append protocol due to multiple failures (§6.2.1).

• **Setup-Stage Recovery:** Finally, Figure 1c shows how the setup-stage recovery is different than the data-transfer recovery. Here, the client first creates a pipeline

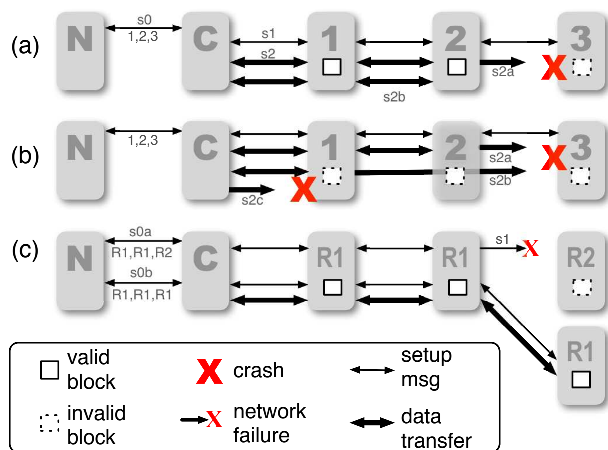


Figure 1: **HDFS Write Recovery Protocol.** *`N`, `C`, `R1/2`, and numeric letters represent the namenode, client, rack number, and datanodes respectively. The client always starts the activity to the namenode first before to the datanodes.*

from two nodes in Rack1 and one in Rack2 (`s0a`). However, due to the rack partitioning (`s1`), the client asks the namenode again for a new fresh pipeline (`s0b`); the client has not transferred any bytes, and thus could start streaming from the beginning. After asking the namenode in several retries (not shown), the pipeline contains only nodes in Rack1 (`s0b`). At the end, all replicas only reside in one rack, which is correct because only one rack is reachable during write [33].

• **Replication Monitor Bug:** Although the previous case is correct, it reveals a crucial design bug in the background replication monitor. This monitor unfortunately only checks the number of replicas but *not* the locations. Thus, even after the partitioning is lifted, the replicas are not migrated to multiple racks. This design bug greatly reduces the block availability if Rack1 is completely unreachable (more in §5.2.3).

To sum up, we have illustrated the complexity of recovery by showing how different failure scenarios lead to different recovery behaviors. There are more problems within this protocol and other protocols. Without an appropriate testing framework, it is hard to ensure recovery correctness; in one discussion of a newly proposed recovery design, a developer raised a comment: "I don't see any proof of correctness. How do we know this will not lead to the same or other problems? [2]"

2.4 Current State of the Art: Does It Help?

In the last three sections, we presented our motivation for powerful testing frameworks for cloud systems. A natural question to ask is whether existing frameworks can help. We answer this question in two parts: failure exploration and system specifications.

2.4.1 Failure Exploration

Developers are accustomed to easy-to-use unit-testing frameworks. For fault-injection purposes, unit tests are severely limited; a unit test often simulates a limited number of failure scenarios, and when it comes to injecting multiple variety of failures, one common practice is to inject a sequence of *random* failures as part of the unit test [6, 35].

To improve common practices, recent work has proposed more exhaustive fault-injection frameworks. For example, the authors of AFEX and LFI observe that the number of possible failure scenarios is "infinite" [20, 28]. Thus, AFEX and LFI automatically prioritize "high-impact targets" (*e.g.*, unchecked system calls, tests likely to fail). So far, they target non-distributed systems and do not address multiple failures in detail.

Recent system model-checkers have also proposed the addition of failures as part of the state exploration strategies [21, 37, 38, 39]. MODIST, for example, is capable of exercising different combinations of failures (*e.g.*, crashes, network failures) [38]. As we discuss later, exploring multiple failures creates a combinatorial explosion problem. This problem has not been addressed by the MODIST authors, and thus they provide a random mode for exploring multiple failures. Overall, we found no work that attempts to systematically explore multiple-failure scenarios, something that cloud systems face more often than other distributed systems in the past [4, 9, 17, 18].

2.4.2 System Specifications

Failure injection addresses only half of the challenge in recovery testing: exercising recovery code. In addition, proper tests require specifications of *expected behavior* from those code paths. In the absence of such specifications, the only behaviors that can be automatically detected are those that interrupt testing (e.g. system failures). One easy way is to write extra checks as part of a unit test. Developers often take this approach, but the problem is there are many specifications to write, and if they are written in imperative languages (*e.g.*, Java) the code is bloated.

Some model checkers use existing consistency checks such as fsck [39], a powerful tool that contains hundreds of consistency checks. However, it has some drawbacks. First, fsck is only powerful if the system is mature enough; developers add more checks across years of development. Second, fsck is also often written in imperative languages, and thus its implementations are complex and unsurprisingly buggy [15]. Finally, fsck can express only "invariant-like" specifications (*i.e.*, it only checks the state of the file system, but not the *events* that lead to the state). As we will see later, specifying recovery requires "behavioral" specifications.

Another advanced checking approach is WiDS [24, 25, 38]. As the target system runs, WiDS interposes and checks the system's internal states. However, it employs a scripting language that still requires a check to be written in tens of lines of code [24, 25]. Furthermore, its interposition mechanism might introduce another issue: the checks are built by interposing specific implementation functions, and if these functions evolve, the checks must be modified. The authors have acknowledged but not addressed this issue [24].

Frameworks for declarative specifications exist (*e.g.*, Pip [32], P2 Monitor [34]). P2 Monitor only works if the target system is written in the same language [34]. Pip facilitates declarative checks, but a check is still written in over 40 lines on average [32]. Also, these systems are not integrated with a failure service, and thus cannot thoroughly test recovery.

Overall, most existing work use approaches that could result in big implementations of the specifications. Managing hundreds of them becomes complicated, and they must also evolve as the system evolves. In practice, developers are reluctant to invest in writing detailed specifications [2], and hence the number of written specifications is typically small.

3 Goals

To address the aforementioned challenges, we present a new testing framework for cloud systems: FATE and DESTINI. We first present our concrete goals here.

• **Target systems and users:** We primarily target cloud systems as they experience a wide variety of failures at a higher rate than any other systems in the past [14]. However, our framework is generic and applies to other distributed systems. Our targets so far are HDFS [33], ZooKeeper [19] and Cassandra [23]. We mainly use HDFS as our example in the paper. In terms of users, we target experienced system developers, with the goal of improving their ability to efficiently generate tests and specifications.

• **Seamless integration:** Our approach requires source code availability. However, for adoptability, our framework should not modify the code base significantly. This is accomplished by leveraging mature interposition technology (*e.g.*, AspectJ). Currently our framework can be integrated to any distributed systems written in Java.

• **Rapid and systematic exploration of failures:** Our framework should help cloud system developers explore multiple-failure scenarios automatically and more systematically than random approaches. However, a complete systematic exploration brings a new challenge: a massive combinatorial explosion of failures, which takes tens of hours to explore. Thus, our testing framework must also be equipped with smart exploration strategies (*e.g.*, prioritizing non-similar failure scenarios first).

• **Numerous detailed recovery specifications:** Ideally, developers should be able to write as many detailed specifications as possible. The more specifications written, the finer bug reports produced, the less time needed for debugging. To realize this, our framework must meet two requirements. First, the specifications must be developer-friendly (*i.e.*, concise, fast to write, yet easy to understand). Otherwise, developers will be reluctant to invest in writing specifications. Second, our framework must facilitate "behavioral" specifications. We note that existing work often focuses on "invariant-like" specifications. This is not adequate because recovery behaves differently under different failure scenarios, and while recovery is still ongoing, the system is likely to go through transient states where some invariants are not satisfied.

4 FATE: Failure Testing Service

Within a distributed execution, there are many points in place and time where system components could fail. Thus, our goal is to exercise failures more methodically than random approaches. To achieve this, we present three contributions: a failure abstraction for expressing failure scenarios (§4.1), a ready-to-use failure service which can be integrated seamlessly to cloud systems (§4.2), and novel failure prioritization strategies that speed up testing time by an order of magnitude (§4.3).

4.1 Failure IDs: Abstraction For Failures

FATE's ultimate goal is to exercise as many combinations of failures as possible. In a sense, this is similar to model checking which explores different sequences of states. One key technique employed in system model checkers is to record the hashes of the explored states. Similarly in our case, we introduce the concept of *failure IDs*, an abstraction for failure scenarios which can be hashed and recorded in history. A failure ID is composed of an I/O ID and the injected failure (Table 1). Below we describe these subcomponents in more detail.

• **I/O points:** To construct a failure ID, we choose I/O points (*i.e.*, system/library calls that perform disk or network I/Os) as failure points, mainly for three reasons. First, hardware failures manifest into failed I/Os. Second, from the perspective of a node in distributed systems, I/O points are critical points that either change its internal states or make a change to its outside world (*e.g.*, disks, other nodes). Finally, I/O points are basic operations in distributed systems, and hence an abstraction built on these points can be used for broader purposes.

• **Static and dynamic information:** For each I/O point, an I/O ID is generated from the static (*e.g.*, system call, source file) and dynamic information (*e.g.*, stack trace, node ID) available at the point. Dynamic information

I/O ID Fields			Values
Static	Func. call	:	OutputStream.flush()
	Source File	:	BlockRecv.java (line 45)
Dynamic	Stack trace	:	(the stack trace)
	Node Id	:	Node2
Domain specific	Source	:	Node2
	Dest.	:	Node1
	Net. Mesg.	:	Setup Ack
Failure ID = hash (I/O ID + Crash) = 2849067135			

Table 1: **A Failure ID.** *A failure ID comprises an I/O ID plus the injected failure (e.g., crash). Hash is used to record a failure ID. For space, some fields are not shown.*

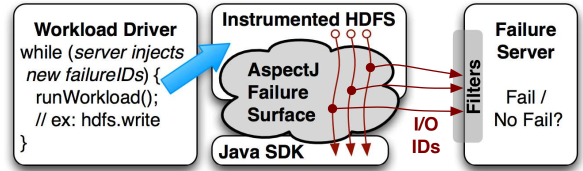


Figure 2: **FATE Architecture.**

is useful to increase failure coverage. For example, recovery might behave differently if a failure happens in different nodes (*e.g.*, first vs. last node in the pipeline).

• **Domain-specific information:** To increase failure coverage further, an I/O ID carries domain-specific information; a common I/O point could write to different file types or send messages to different nodes. FATE's interposition mechanism provides runtime information available at an I/O point such as the target I/O (*e.g.*, file names, IP addresses) and the I/O buffer (*e.g.*, network packet, file buffer). To convert these raw information into a more meaningful context (*e.g.*, "Setup Ack" in Table 1), FATE provides an interface that developers can implement. For example, given an I/O buffer of a network message, a developer can implement the code that reverse-engineers the byte content of the message into a more meaningful message type (*e.g.*, "Setup Ack"). If the interface is empty, FATE can still run (the interface returns an empty domain-specific string), but failure coverage could be sacrificed.

• **Possible failure modes:** Given an I/O ID, FATE generates a list of possible failures that could happen on the I/O. For example, FATE could inject a disk failure on a disk write, or a network failure before a node sends a message. Currently, we support six failure types: crash, permanent disk failure, disk corruption, node-level and rack-level network partitioning, and transient failure. To create a failure ID, one failure type appropriate to the I/O is selected one at a time (and hence, given an I/O ID, FATE could produce multiple failure IDs).

4.2 Architecture

We built FATE with a goal of quick and seamless integration into our target systems. Figure 2 depicts the four components of FATE: workload driver, failure surface, failure server, and filters.

4.2.1 Workload Driver, Failure Surface, and Server

We first instrument the target system (*e.g.*, HDFS) by inserting a "failure surface". There are many possible layers to insert a failure surface (*e.g.*, inside a system library or at the VMM layer). We do this between the target system and the OS library (*e.g.*, Java SDK), for two reasons. First, at this layer, rich domain-specific information is available. Second, by leveraging mature instrumentation technology (*e.g.*, AspectJ), adding the surface requires no modification to the code base.

The failure surface has two important jobs. First, at each I/O point, it builds the I/O ID. Second, it needs to check if a persistent failure injected in the past affects this I/O point (*e.g.*, network partitioning). If so, the surface returns an error to emulate the failure without the need to talk to the server. Otherwise, it sends the I/O ID to the server and receives a failure decision.

The workload driver is where the developer attaches the workload to be tested (*e.g.*, write, append, or some sequence of operations, including the pre- and post-setups) and specifies the maximum number of failures injected per run. As the workload runs, the failure server receives I/O IDs from the failure surface, combines the I/O IDs with possible failures into failure IDs, and makes failure decisions based on the failure history. The workload driver terminates when the server does not inject a new failure scenario. The failure server, workload driver, and target system are run as separate processes, and they can be run on single or multiple machines.

4.2.2 Brute-Force Failure Exploration

By default, FATE runs in brute-force mode. That is, FATE systematically explores all possible combinations of observed failure IDs. (The algorithm can be found in our technical report [13]). With this brute-force mode, FATE has exercised over 40,000 *unique* combinations of one, two and three failure IDs. We address this combinatorial explosion challenge in the next section (§4.3).

4.2.3 Filters

FATE uses information carried in I/O and failure IDs to implement filters at the server side. A filter can be used to regenerate a particular failure scenario. For example, to regenerate the failure described in Table 1, a developer could specify a filter that will only exercise the corresponding failure ID. A filter could also be used to reduce the failure space. For example, a developer could insert a filter that allows crash-only failures, failures only on some specific I/Os, or any failures only at datanodes.

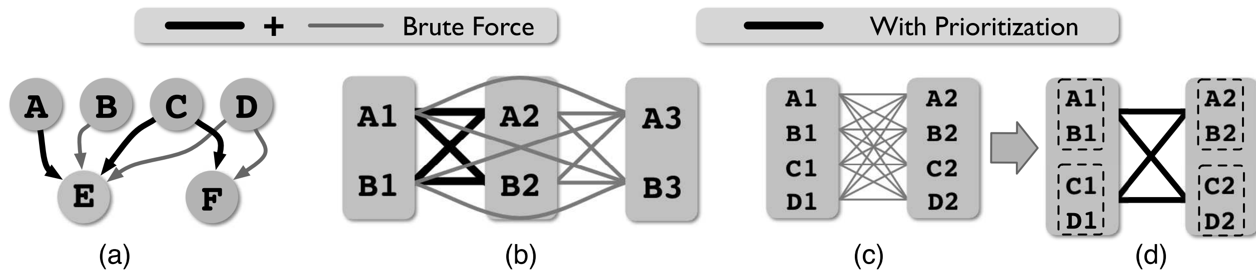


Figure 3: **Prioritization of Pairwise Dependent and Independent Failures.**

4.3 Failure Exploration Strategy

Running FATE in brute-force mode is impractical and time consuming. As an example, we have run the append protocol with a filter that allows crash-only failures on disk I/Os in datanodes. With this filter, injecting two failures per run gives 45 failure IDs to exercise, which leads us to 1199 combinations that take more than 2 hours to run. Without the filter (*i.e.*, including network I/Os and other types of failures) the number will further increase. This introduces the problem of exponential explosion of multiple failures, which has to be addressed given the fact that we are dealing with large code base where an experiment could take more than 5 seconds per run (*e.g.*, due to pre- and post-setup overheads).

Among the 1199 experiments, 116 failed; if recovery is perfect, all experiments should be successful. Debugging all of them led us to 3 bugs as the root causes. Now, we can concretely define the challenge: *Can* FATE *exercise a much smaller number of combinations and find distinct bugs faster?* This section provides some solutions to this challenge. To the best of our knowledge, we are the first to address this issue in the context of distributed systems. Thus, we also hope that this challenge attracts system researches to present other alternatives.

To address this challenge, we have studied the properties of multiple failures (for simplicity, we begin with two-failure scenarios). A pair of two failures can be categorized into two types: *pairwise dependent* and *pairwise independent* failures. Below, we describe each category along with the prioritization strategies. Due to space constraints, we could not show the detailed pseudo-code, and thus we only present the algorithms at a high-level. We will evaluate the algorithms in Section 6.3. We also emphasize that our proposed strategies are built on top of the information carried in failure IDs, and hence display the power of failure IDs abstraction.

4.3.1 Pairwise Dependent Failures

A pair of failure IDs is dependent if the second ID is *observed* only if the failure on the first ID is *injected*; observing the occurrence of a failure ID does not necessarily mean that the failure must be injected. The key here is to use observed I/Os to capture path coverage information (this is an acceptable assumption since we are dealing with distributed systems where recovery essentially manifests into I/Os). Figure 3a illustrates some combinations of dependent failure IDs. For example, `F` is dependent on `C` or `D` (*i.e.*, `F` will never be observed unless `C` or `D` is injected). The brute-force algorithm will inefficiently exercise all six possible combinations: `AE`, `BE`, `CE`, `DE`, `CF`, and `DF`.

To prioritize dependent failure IDs, we introduce a strategy that we call *recovery-behavior clustering*. The goal is to prioritize "non-similar" failure scenarios first. The intuition is that non-similar failure scenarios typically lead to different recovery behaviors, and recovery behaviors can be represented as a sequence of failure IDs. Thus, to perform the clustering, we first run a complete set of experiments with *only one* failure per run, and in each run we record the *subsequent* failure IDs.

We formally define subsequent failure IDs as all observed IDs after the injected failure up to the point where the system enters the *stable state*. That is, recording recovery only up to the end of the protocol (*e.g.*, write) is not enough. This is because a failed I/O could leave some "garbage" that is only cleaned up by some background protocols. For example, a failed I/O could leave a block with an old generation timestamp that should be cleaned up by the background replication monitor (outside the scope of the write protocol). Moreover, different failures could leave different types of garbage, and thus lead to different recovery behaviors of the background protocols. By capturing subsequent failure IDs until the stable state, we ensure more fine-grained clustering.

The exact definition of stable state might be different across different systems. For HDFS, our definition of stable state is: FATE reboots dead nodes if any, removes transient failures (*e.g.*, network partitioning), sends commands to the datanodes to report their blocks to the namenode, and waits until all datanodes receive a null command (*i.e.*, no background jobs to run).

Going back to Figure 3a, the created mappings between the first failures and their subsequent failure IDs are: {`A`→ `E`}, {`B`→ `E`}, {`C`→ `E`, `F`}, and {`D`→ `E`, `F`}. The recovery behaviors then are clustered into two: {`E`}, and {`E`, `F`}. Finally, for each recovery cluster, we pick only

one failure ID on which the cluster is dependent. The final prioritized combinations are marked with bold edges in Figure 3a. That is, FATE only exercises: AE, CE, and CF. Note that E is exercised as a second failure twice because it appears in different recovery clusters.

4.3.2 Pairwise Independent Failures

A pair of failure IDs is independent if the second ID is observed even if the first ID is *not* injected. This case is often observed when the same piece of code runs in parallel, which is a common characteristic found in distributed systems (*e.g.*, two phase commit, leader election, HDFS write and append). Figure 3b illustrates a scenario where the same I/O points A and B are executed concurrently in three nodes (*i.e.*, A1, A2, A3, B1, B2, B3). Let's name these two I/O points A and B as static failure points, or SFP in short (as they exclude node ID). With brute-force exploration, FATE produces 24 combinations (the 12 bi-directional edges in Figure 3b). In more general, there are $SFP^2 * N(N-1)$ combinations, where N and SFP are the number of nodes and static failure points respectively. To reduce this quadratic growth, we introduce two levels of prioritization: one for reducing $N(N-1)$ and the other for SFP^2.

To reduce $N(N-1)$, we leverage the property of *symmetric code* (*i.e.*, the same code that runs concurrently in different nodes). Because of this property, if a pair of failures has been exercised at two static failure points of two specific nodes, it is not necessary to exercise the same pair for other pairs of nodes. For example, if A1B2 has been exercised, it is not necessary to run A1B3, A2B1, A2B3, and so on. As a result, we have reduced $N(N-1)$ (*i.e.*, any combinations of two nodes) to just one (*i.e.*, a pair of two nodes); the N does not matter anymore.

Although the first level of reduction is significant, FATE still hits the SFP^2 bottleneck as illustrated in Figure 3c. Here, instead of having two static failure points, there are four, which leads to 16 combinations. To reduce SFP^2, we utilize the behavior clustering algorithm used in the dependent case. That is, if injecting failure ID A1 results in the same recovery behavior as in injecting B1, then we cluster them together (*i.e.*, only one of them needs to be exercised). Put simply, the goal is to reduce SFP to $SFP_{clustered}$, which will reduce the input to the quadratic explosion (*e.g.*, from 4 to 2 resulting in 4 uni-directional edges as depicted in Figure 3d). In practice, we have seen a reduction from fifteen SFP to eight $SFP_{clustered}$.

4.4 Summary

We have introduced FATE, a failure testing service capable of exploring multiple, diverse failures in systematic fashion. FATE employs failure IDs as a new abstraction for exploring failures. FATE is also equipped with prioritization strategies that prioritize failure scenarios that result in distinct recovery actions. Our approaches are not sound; however by experience, all bugs found with brute-force are also found with prioritization (more in §6.3). If developers have the time and resources, they could fall back to brute-force mode for more confidence. So far, we have only explained our algorithms for two-failure scenarios. We have generalized them to three-failure, but cannot present them due to space constraints. One fundamental limitation of FATE is the absence of I/O reordering [38], and thus it is possible that some orderings of failures are not exercised. Adopting related techniques from existing work [38] will be be beneficial in our case.

5 DESTINI: Declarative Testing Specifications

After failures are injected, developers still need to check for system correctness. As described in the motivation (§2.4), DESTINI attempts to improve the state-of-the-art of writing system specifications. In the following sections, we first describe the architecture (§5.1), then present some examples (§5.2), and finally summarize the advantages (§5.3). Currently, we target recovery bugs that reduce availability (*e.g.*, unmasked failures, fail-stop) and reliability (*e.g.*, data-loss, inconsistency). We leave performance and scalability bugs for future work.

5.1 Architecture

At the heart of DESTINI is Datalog, a declarative relational logic language. We chose the Datalog style as it has been successfully used for building distributed systems [3, 26] and for verifying some aspects of system correctness (*e.g.*, security [12, 31]). Unlike much of that work, we are not using Datalog to implement system internals, but only to write correctness specifications that are checked relatively rarely. Hence we are less dependent on the efficiency of current Datalog engines, which are still evolving [3].

In terms of the architecture, DESTINI is designed such that developers can build specifications from minimal information. To support this, DESTINI comprises three features as depicted in Figure 4. First, it interposes network and disk protocols and translates the available information into Datalog events (*e.g.*, *cnpEv*). Second, it records failure scenarios by having FATE inform DESTINI about failure events (*e.g.*, *fateEv*). This highlights that FATE and DESTINI must work hand in hand, a valuable property that is apparent throughout our examples. Finally, based *only* on events, it records facts, deduces expectations of how the system should behave in the future, and

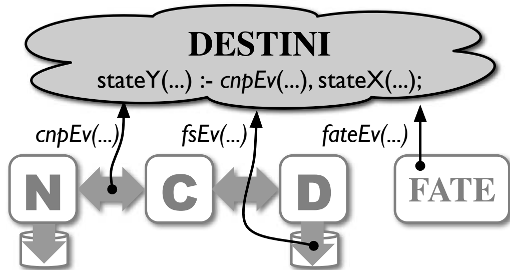


Figure 4: **DESTINI Architecture.**

compares the two.

5.1.1 Rule Syntax

In DESTINI, specifications are formally written as Datalog rules. A rule is essentially a logical relation:

```
errX(P1,P2,P3) :- cnpEv(P1), NOT-IN stateY(P1,P2,_),
                  P2 == img, P3 := Util.strLib(P2);
```

This Datalog rule consists of a head table (`errX`) and predicate tables in the body (*`cnpEv`* and `stateY`). The head is evaluated when the body is true. Tuple variables begin with an upper-case letter (`P1`). A don't care variable is represented with an underscore (_). A comma between predicates represents conjunction. "`:=`" is for assignments. We also provide some helper libraries (`Util.strLib()` to manipulate strings). Lower case variables (`img`) represent integer or string constants. All upper case letters (`NOT-IN`) are Datalog keywords. Events are in italic. To help readers track where events originate from, an event name begins with one of these labels: *`cnp`*, *`dnp`*, *`cdp`*, *`ddp`*, *`fs`*, which stand for client-namenode, datanode-namenode, client-datanode, datanode-datanode, and file system protocols respectively (Figure 4). Non-event (non-italic) heads and predicates are essentially database tables with primary keys defined in some schemas (not shown). A table that starts with `err` represents an error (*i.e.*, if a specification is broken, the error table is non-empty, implying the existence of one or more bugs).

5.2 DESTINI Examples

This section presents the powerful features of DESTINI via four examples of HDFS recovery specifications. In the first example, we present five important components of recovery specifications (§5.2.1). To help simplify the complex debugging process, the second example shows how developers can incrementally add tighter specifications (§5.2.2). The third example presents specifications that incorporate a different type of failure than the first two examples (§5.2.3). Finally, we illustrate how developers can refine existing specifications (§5.2.4).

5.2.1 Specifying Data-Transfer Recovery

DESTINI facilitates five important elements of recovery specifications: checks, expectations, facts, precise failure events, and check timings. Here, we present these elements by specifying the data-transfer recovery protocol (Figure 1a); this recovery is correct if valid replicas are stored in the surviving nodes of the pipeline.

• **Checks:** To catch violations of data-transfer recovery, we start with a simple high-level *check* (**a1**), which says "upon block completion, throw an error if there is a node that is expected to store a valid replica, but actually does not." This rule shows how a check is composed of three elements: the *expectation* (`expectedNodes`), *fact* (`actualNodes`), and *check timing* (*`cnpComplete`*).

• **Expectations:** The expectation (`expectedNodes`) is deduced from protocol events (**a2**-**a8**). First, without any failure, the expectation is to have the replicas in all the nodes in the pipeline (**a3**); information about pipeline nodes are accessible from the setup reply from the namenode to the client (**a2**). However, if there is a crash, the expectation changes: the crashed node should be removed from the expected nodes (**a4**). This implies that an expectation is also based on *failure events*.

• **Failure events:** Failures in different stages result in different recovery behaviors. Thus, we must know precisely when failures occur. For data-transfer recovery, we need to capture the current stage of the write process and only change the expectation if a crash occurs within the data-transfer stage (*`fateCrashNode`* happens at `Stg==2` in rule **a4**). The data transfer stage is deduced in rules **a5**-**a8**: the second stage begins after all acks from the setup phase have been received.

Before moving on, we emphasize two important observations here. First, this example shows how FATE and DESTINI must work hand in hand. That is, recovery specifications require a failure service to exercise them, and a failure service requires specifications of expected failure handling. Second, with logic programming, developers can easily build expectations only from events.

• **Facts:** The fact (`actualNodes`) is also built from events (**a9**-**a16**); specifically, by tracking the locations of valid replicas. A valid replica can be tracked with two pieces of information: the block's latest generation time stamp, which DESTINI tracks by interposing two interfaces (**a9** and **a10**), and meta/checksum files with the latest generation timestamp, which are obtainable from file operations (**a11**-**a15**). With this information, we can build the runtime fact: the nodes that store the valid replicas of the block (**a16**).

• **Check timings:** The final step is to compare the expectation and the fact. We underline that the timing of the check is important because we are specifying *recovery behaviors*, unlike invariants which must be true at

	Section 5.2.1		Data-Transfer Recovery Specifications
a1	errDataRec (B, N)	:-	*cnpComplete* (B), expectedNodes (B, N), NOT-IN actualNodes (B, N);
a2	pipeNodes (B, Pos, N)	:-	*cnpGetBlkPipe* (UFile, B, Gs, Pos, N);
a3	expectedNodes (B, N)	:-	pipeNodes (B, Pos, N);
a4	DEL expectedNodes (B, N)	:-	*fateCrashNode* (N), pipeStage (B, Stg), Stg == 2, expectedNodes (B, N);
a5	setupAcks (B, Pos, Ack)	:-	*cdpSetupAck* (B, Pos, Ack);
a6	goodAcksCnt (B, COUNT<Ack>)	:-	setupAcks (B, Pos, Ack), Ack == 'OK';
a7	nodesCnt (B, COUNT<Node>)	:-	pipeNodes (B, _, N, _);
a8	pipeStage (B, Stg)	:-	nodesCnt (NCnt), goodAcksCnt (ACnt), NCnt == Acnt, Stg := 2;
a9	blkGenStamp (B, Gs)	:-	*dnpNextGenStamp* (B, Gs);
a10	blkGenStamp (B, Gs)	:-	*cnpGetBlkPipe* (UFile, B, Gs, _, _);
a11	diskFiles (N, File)	:-	*fsCreate* (N, File);
a12	diskFiles (N, Dst)	:-	*fsRename* (N, Src, Dst), diskFiles (N, Src);
a13	DEL diskFiles (N, Src)	:-	*fsRename* (N, Src, Dst), diskFiles (N, Src);
a14	fileTypes (N, File, Type)	:-	diskFiles(N, File), Type := Util.getType(File);
a15	blkMetas (N, B, Gs)	:-	fileTypes (N, File, Type), Type == metafile, B := Util.getBlk(File), Gs := Util.getGs(File);
a16	actualNodes (B, N)	:-	blkMetas (N, B, Gs), blkGenStamp (B, Gs);

	Section 5.2.2		Tighter Specifications for Data-Transfer Recovery
b1	errBadAck (Pos, N)	:-	*cdpDataAck* (Pos, 'Error'), pipeNodes (B, Pos, N), liveNodes (N);
b2	liveNodes (N)	:-	*dnpRegistration* (N);
b3	DEL liveNodes (N)	:-	*fateCrashNode* (N);
b4	errBadConnect (N, TgtN)	:-	*ddpDataTransfer* (N, TgtN, Status), liveNodes (TgtN), Status == terminated;

	Section 5.2.3		Rack-Aware Policy Specifications
c1	warnSingleRack (B)	:-	rackCnt (B, 1), actualRacks (B, R), connectedRacks (R, OtherR);
c2	actualRacks (B, R)	:-	actualNodes (B, N), nodeRackMap (N, R);
c3	rackCnt (B, COUNT<R>)	:-	actualRacks (B, R);
c4	DEL connectedRacks (R1, R2)	:-	fatePartitionRacks (R1, R2);
c5	err1RackOnCompletion (B)	:-	*cnpComplete* (B), warnSingleRack (B);
c6	err1RackOnStableState (B)	:-	*fateStableState* (_), warnSingleRack (B);

	Section 5.2.4		Refining Log-Recovery Specifications
d1	errLostUFile (UFile)	:-	expectedUFile (UFile), NOT-IN ufileInNameNode (UFile);
d2	ufileInNameNode (UFile) **	:-	ufileInNnFile(F, NnFile), (NnFile == img || NnFile == log || NnFile == img2);
d3	ufileInNameNode (UFile)	:-	ufileInNnFile (F, img2), logRecStage (Stg), Stg == 4;
d4	ufileInNameNode (UFile)	:-	ufileInNnFile (F, img) , logRecStage (Stg), Stg != 4;
d5	ufileInNameNode (UFile)	:-	ufileInNnFile (F, log) , logRecStage (Stg), Stg != 4;

Table 2: **Sample Specifications.** *The table lists all the rules we wrote to specify the problems in Section 5.2; Rules* `aX`, `bX`, `cX`, *and* `dX` *are for Sections 5.2.1, 5.2.2, 5.2.3, and 5.2.4 respectively. All logical relations are built only from events (in italic). The shaded rows indicate checks that catch violations. A check always starts with* `err`. *Tuple variables* `B`, `Gs`, `N`, `Pos`, `R`, `Stg`, `NnFile`, *and* `UFile` *are abbreviations for block, generation timestamp, node, position, rack, stage, namenode file, and user file respectively; others should be self-explanatory. Each table has primary keys defined in a schema (not shown). (**) Rule* `d2` *is refined in* `d3` *to* `d5`*; these rules are described more in our short paper [14].*

all time. Not paying attention to this will result in false warnings (*i.e.*, there is a period of time when recovery is ongoing and specifications are not met). Thus, we need precise events to signal check times. In this example, the check time is at block completion (*cnpComplete* in **a1**).

5.2.2 Debugging with Tighter Specifications

The rules in the previous section capture the high-level objective of HDFS data-transfer recovery. After we ran FATE to cover the first crash scenario in Figure 1b (for simplicity of explanation, we exclude the second crash), rule **a1** throws an error due to a bug that wrongly excludes the good second node (Figure 1b in §2.3). Although the check unearths the bug, it does not *pinpoint* the bug (*i.e.*, answer *why* the violation is thrown).

To improve this debugging process, we added more detailed specifications. In particular, from the events that DESTINI logs, we observed that the client excludes the second node in the next pipeline, which is possible if the client receives a bad ack. Thus, we wrote another check (**b1**) which says "throw an error if the client receives a bad ack for a live node" (**b1**'s predicates are specified in **b2** and **b3**). Note that this check is written from the *client's view*, while rule **a1** from the *global view*.

The new check catches the bug closer to the source, but also raises a new question: Why does the client receive a bad ack for the second node? One logical explanation is because the first node cannot communicate to the second node. Thus, we easily added many checks that catch unexpected bad connections such as **b4**, which finally pinpoints the bug: the second node, upon seeing a failed connection to the crashed third node, incorrectly closes the streams connected to the first node; note that this check is written from the *datanode's view*.

In summary, more detailed specifications prove to be valuable for assisting developers with the complex debugging process. This is unlikely to happen if a check implementation is long. But with DESTINI, a check can be expressed naturally in a small number of logical relations. Moreover, checks can be written from different views (*e.g.*, global, client and datanode as shown in **a1**, **b1**, **b4** respectively). Table 3 shows a timeline of when these various checks are violated. As shown, tighter specifications essentially fill the "explanation gaps" between the injected failure and the wrong final state of the system.

5.2.3 Specifying Rack-Aware Replication Policy

In this example, we write specifications for the HDFS rack-aware replication policy, an important policy for high availability [10, 33]. Unlike previous examples, this example incorporates network partitioning failure mode.

According to the HDFS architects [33], the write protocol should ensure that block replicas are spread across a minimum of two available racks. But, if only one rack is reachable, it is acceptable to use one rack temporarily. To express this, rule **c1** throws a warning if a block's rack could reach another rack, but the block's rack count is one (rules **c2**-**c4** provide topology information, which is initialized when the cluster starts and updated when FATE creates a rack partition). This warning becomes a hard error *only* if it is true upon block completion (**c5**) or stable state (**c6**). Note again how these timings are important to prevent false errors; while recovery is ongoing, replicas are still being re-shuffled into multiple racks.

With these checks, DESTINI found the bug in Figure 1c (§2.3), a critical bug that could greatly reduce availability: all replicas of a block are stored in a single rack. Note that the bug does not violate the completion rule (because the racks are still partitioned). But, it does violate the stable state rule because even after the network partitioning is removed, the replication monitor does not re-shuffle the replicas.

Time, Events, and Errors
t1: Client asks the namenode for a block ID and the nodes.
cnpGetBlkPipe (usrFile, blk_x, gs1, 1, N1);
cnpGetBlkPipe (usrFile, blk_x, gs1, 2, N2);
cnpGetBlkPipe (usrFile, blk_x, gs1, 3, N3);
t2: Setup stage begins (pipeline nodes setup the files). *
fsCreate (N1, tmp/blk_x_gs1.meta);
fsCreate (N2, tmp/blk_x_gs1.meta);
fsCreate (N3, tmp/blk_x_gs1.meta);
t3: Client receives setup acks. Data transfer begins.
cdpSetupAck (blk_x, 1, OK);
cdpSetupAck (blk_x, 2, OK);
cdpSetupAck (blk_x, 3, OK);
t4: FATE crashes N3. **Got error** (b4).
fateCrashNode (N3);
errBadConnect (N1, N2); // should be good
t5: Client receives an erroneous ack. **Got error** (b1).
cdpDataAck (2, Error);
errBadAck (2, N2); // should be good
t6: Recovery begins. Get new generation time stamp.
dnpNextGenStamp (blk_x, gs2);
t7: Only N1 continues and finalizes the files.
fsCreate (N1, tmp/blk_x_gs2.meta);
fsRename (N1, tmp/blk_x_gs2.meta, current/blk_x_gs2.meta);
t8: Client marks completion. **Got error** (a1).
cnpComplete (blk_x);
errDataRec (blk_x, N2); // should exist

Table 3: **A Timeline of DESTINI Execution.** *The table shows the timeline of runtime events (italic) and errors (shaded). Tighter specifications capture the bug earlier in time. The tuples (strings/integers) are real entries (not variable names). For space, we do not show block-file creations (but only meta files*) nor how the rules in Table 2 are populated.*

5.2.4 Refining Specifications

In the second example (§5.2.2), we demonstrated how developers can *incrementally add* detailed specifications. In this section, we briefly show how developers can *refine* existing specifications (an extensive description can be found in our short paper [14]).

Here, we specify the HDFS log-recovery process in order to catch data-loss bugs in this protocol. The high-level check (**d1**) is fairly simple: "a user file is lost if it does not exist at the namenode." To capture the facts, we wrote rule **d2** which says "*at any time*, user files should exist in the union of all the three namenode files used in log recovery." With these rules, we found a data-loss bug that accidentally deletes the metadata of user files. But, the error is only thrown *at the end* of the log recovery process (*i.e.*, the rules are not detailed enough to pinpoint the bug). We then refined rule **d2** to reflect in detail the four stages of the process (**d3** to **d5**). That is, depending on the stage, user files are expected to be in a different subset of the three files. With these refined specifications, the data-loss bug was captured in between stage 3 and 4.

5.3 Summary of Advantages

Throughout the examples, we have shown the advantages of DESTINI: it facilitates checks, expectations, facts, failure events, and precise timings; specifications can be written from different views (*e.g.*, global, client, datanode); different types of violations can be specified (*e.g.*, availability, data-loss); different types of failures can be incorporated (*e.g.*, crashes, partitioning); and specifications can be incrementally added or refined. Overall, the resulting specifications are clear, concise, and precise, which potentially attracts developers to write many specifications to ease complex debugging process. All of these are feasible due to three important properties of DESTINI: the interposition mechanism that translates disk and network events; the use of relational logic language which enables us to deduce complex states only from events; and the inclusion of failure events from the collaboration with FATE. Besides these advantages, adopting DESTINI requires one major effort: developers need to reverse-engineer raw I/O information (*e.g.*, I/O buffer, stack trace) collected from the Java-based interposition mechanism into semantically-richer Datalog events (*e.g.*, `cnpComplete`). However, we hope that this effort will also be useful for other debugging techniques that need detailed I/O information.

6 Evaluation

We evaluate FATE and DESTINI in several aspects: the general usability for cloud systems (§6.1), the ability to catch multiple-failure bugs (§6.2), the efficiency of our prioritization strategies (§6.3), the number of specifications we have written and their reusability (§6.4), the number of new bugs we have found and old bugs reproduced (§6.5), and the implementation complexity (§6.6).

Since we currently only test reliability (but not performance), it is sufficient to run FATE, DESTINI, and the target systems as separate processes on a single machine; network and disk failures are emulated (manifested as Java I/O exceptions), and crashes are emulated with process crashes. Nevertheless, FATE and DESTINI can run on separate machines.

6.1 Target Systems and Protocols

We have integrated FATE and DESTINI to three cloud systems: HDFS [33] v0.20.0 and v0.20.2+320 (the latter is released in Feb. 2010 and used by Cloudera and Facebook), ZooKeeper [19] v3.2.2 (Dec. 2009), and Cassandra [23] v0.6.1 (Apr. 2010). We have run our framework on four HDFS workloads (log recovery, write, append, and replication monitor), one ZooKeeper workload (leader election), and one Cassandra workload (key-value insert). In this paper, we only present extensive evaluation numbers for HDFS. For Cassandra and ZooKeeper, we only present partial results.

6.2 Multiple-Failure Bugs

The uniqueness of our framework is the ability to explore multiple failures systematically, and thus catch corner-case multiple-failure bugs. Here, we describe two out of five multiple-failure bugs that we found.

6.2.1 Append Bugs

We begin with a multiple-failure bug in the HDFS append protocol. Unlike write, append is more complex because it must atomically mutate block replicas [36]. HDFS developers implement append with a custom protocol; their latest append design was written in a 19-page document of prose specifications [22]. Append was finally supported after being a top user demand for three years [36]. As a note, Google FS also supports append, but its authors did not share their internal design [10].

In the experiment setup, a block has three replicas in three nodes, and thus should survive two failures. On append, the three nodes form a pipeline. N1 starts a thread that streams the new bytes to N2 and then N1 appends the bytes to its block. N2 crashes at this point, and N1 sends a bad ack to the client, but does not stop the thread. Before the client continues streaming via a new pipeline, all surviving nodes (N1 and N3) must agree on the same block offset (the `syncOffset` process). In this process, each node stops the writing thread, verifies that the block's in-memory and on-disk lengths are the same,

broadcasts the offset, and picks the smallest offset. However, N1 might have not updated the block's in-memory length, and thus throws an exception resulting in the new pipeline containing only N3. Then, N3 crashes, and the pipeline is empty. The append fails, but worse, the block in N1 (still alive) becomes "trapped" (*i.e.*, inaccessible). After FATE ran all the background protocols (*e.g.*, lease recovery), the block is still trapped and permanently inaccessible. We have submitted a fix for this bug [2].

6.2.2 Combinations of Different Failures

We have also found a new data-loss bug due to a sequence of *different* failure modes, more specifically, transient disk failure (#1), crash (#2), and disk corruption (#3) at the namenode. The experiment setup was that the namenode has three replicas of metadata files on three disks, and one disk is flaky (exhibits transient failures and corruptions). When users store new files, the namenode logs them to all the disks. If a disk (*e.g.*, Disk1) returns a transient write error (#1), the namenode will exclude this disk; future writes will be logged to the other two disks (*i.e.*, Disk1 will contain stale data). Then, the namenode crashes after several updates (#2). When the namenode reboots, it will load metadata from the disk that has the latest update time. Unfortunately, the file that carries this information is not protected by a checksum. Thus, if this file is corrupted (#3) such that the update time of Disk1 becomes more recent than the other two, then the namenode will load stale data, and flush the stale data to the other two disks, wiping out all recent updates. One could argue that this case is rare, but cloud-scale deployments cause rare bugs to surface; a similar case of corruption did occur in practice [2]. Moreover, data-loss bugs are serious ones [27, 29, 30].

6.3 Prioritization Efficiency

When FATE was first deployed without prioritization, we exercised over 40,000 unique combinations of failures, which combine into 80-hour of testing time. Thousands of experiments failed (probably only due to tens of bugs). Although 80 hours seems a reasonable testing time to unearth crucial reliability bugs, this long testing time only covers several workloads; in reality, there are more workloads to test. In addition, as developers modify their code, they likely to prefer faster turn-around time to find new bugs from their new changes. Overall, this long testing is an overwhelming situation, but which fortunately unfolds into a good outcome: new strategies for multiple-failure prioritization.

To evaluate our strategies, we first focused only on two protocols (write and append) because we need to compare the brute-force with the prioritization results. More specifically, for each method, we count the number of combinations and the number of distinct bugs. Our hope

Workload	#F	STR	#EXP	FAIL	BUGS
Append	2	BF	**1199**	116	**3**
		PR	**112**	17	**3**
Append	3	BF	**7720**	**3693	***3**
		PR	**618**	72	***3**
Write	2	BF	**524**	120	**2**
		PR	**49**	27	**2**
Write	3	BF	**3221**	911	***2**
		PR	**333**	82	***2**

Table 4: **Prioritization Efficiency.** *The columns from left to right are the number of injected failures per run (F), exploration strategy (STR), combinations/experiments (EXP), failed experiments (FAIL), and bugs found (BUGS). BF and PR stands for brute-force and prioritization respectively. Note that the bug counts are only due to two and three failures and depend on the filter (i.e., there are more bugs than shown). (*) Bugs in three-failure experiments are the same as in two-failure ones. (**) This high number is due to a design bug; we used triaging to help us classify the bugs (not shown).*

is that the latter is the same for brute-force and prioritization. Table 4 shows the result of running the two workloads with two and three failures per run, and with a lightweight filter (crash-only failures on disk I/Os in datanodes); without this filter, the number of brute-force experiments is too large to debug. In short, the table shows that our prioritization strategies reduce the total number of experiments by an order of magnitude (the testing time for the workloads in Table 4 is reduced from 26 hours to 2.5 hours). In addition, from our experience no bugs are missing. Again, we cannot prove that our approach is sound; developers could fall back to brute-force for more confidence. Table 4 also highlights the exponential explosion of combinations of multiple failures; the numbers for three failures are much higher than those for two failures (*e.g.*, 7720 vs. 1119). So far, we only cover up to 3 failures, and our techniques still scale reasonably well (*i.e.*, they still give an order of magnitude improvement).

6.4 Specifications

In the last six months, we have written 74 checks on top of 174 rules for a total of 351 lines (65 checks for HDFS, 2 for ZooKeeper, and 7 for Cassandra). We want to emphasize that $\frac{rules}{checks}$ ratio displays how DESTINI empowers specification reuse (*i.e.*, building more checks on top of existing rules). As a comparison, the ratio for our first check (§5.2.1 in Table 2) is 16:1, but the ratio now is 3:1.

Table 5 compares DESTINI with other related work. The table highlights that DESTINI allows a large number of checks to be written in fewer lines of code. We want to note that the number of specifications we have written so far only represents six recovery protocols; there are more that can be specified. As time progresses, we believe the

Type	Framework	#Chks	Lines/Chk
S/I	D3S [24]	10	**53**
D/I	Pip [32]	44	**43**
S/I	WiDS [25]	15	**22**
D/D	P2 Monitor [34]	11	**12**
D/I	DESTINI	74	**5**

Table 5: **DESTINI vs. Related Work.** *The table compares* DESTINI *with related work. D, S, and I represent declarative, scripting, and imperative languages respectively. X/Y implies specifications in X language for systems in Y language. We divide existing work into three classes (S/I, D/D, D/I).*

simplicity offered by DESTINI will open the possibility of having hundreds of specifications along with more recovery specification patterns.

To show how our style of writing specifications is applicable to other systems, we present in more detail some specifications we wrote for ZooKeeper and Cassandra.

6.4.1 ZooKeeper

We have integrated our framework to ZooKeeper [19]. We picked two reported bugs in the version we analyzed. Let's say three nodes N1, N2, and N3, participate in a leader election, and $id(N1) < id(N2) < id(N3)$. If N3 crashes at any point in this process, the expected behavior is to have N1 and N2 form a 2-quorum. However, there is a bug that does not anticipate N3 crashing at a particular point, which causes N1 and N2 to continue nominating N3 in ever-increasing rounds. As a result, the election process never terminates and the cluster never becomes available. To catch this bug, we wrote an invariant violation "a node chooses a winner of a round without ensuring that the chosen leader has in itself voted in the round." The other bug involves multiple failures and can be caught with an addition of just one check; we reuse rules from the first bug. So far, we have written 12 rules for ZooKeeper.

6.4.2 Cassandra

We have also done the same for Cassandra [23], and picked three reported bugs in the version we analyzed. In Cassandra, the key-value insert protocol allows users to specify a consistency level such as `one`, `quorum`, or `all`, which ensures that the client waits until the key-value has been flushed on at least one, N/2 + 1, or all N nodes respectively. These are simple specifications, but again, due to complex implementation, bugs exist and break the rules. For example, at level `all`, Cassandra could incorrectly return a success even when only one replica has been completed. FATE is able to reproduce the failure scenarios and DESTINI is equipped with 7 checks (in 12 rules) to catch consistency-level related bugs.

6.5 New Bugs and Old Bugs Reproduced

We have tested HDFS for over eight months and submitted 16 new bugs, out of which 7 uncovered design bugs (*i.e.*, require protocol modifications) and 9 uncovered implementation bugs. All have been confirmed by the developers. For Cassandra and ZooKeeper, we observed some failed experiments, but since we do not have the chance to debug all of them, we have no new bugs to report.

To further show the power of our framework, we address two challenges: Can FATE reproduce all the failure scenarios of old bugs? Can DESTINI facilitate specifications that catch the bugs? Before proposing our framework for catching unknown bugs, we wanted to feel confident that it is expressive enough to capture known bugs. We went through the 91 HDFS recovery issues (§2.2) and selected 74 that relate to our target workloads (§6.1). FATE is able to reproduce all of them; as a proof, we have created 22 filters (155 lines in Java) to reproduce all the scenarios. Furthermore, we have written checks that could catch 46 old bugs; since some of the old bugs have been fixed in the version we analyzed, we introduced artificial bugs to test our specifications. For ZooKeeper and Cassandra, we have reproduced a total of five bugs.

6.6 FATE and DESTINI Complexity

FATE comprises generic (workload driver, failure server, failure surface) and domain-specific parts (workload driver, I/O IDs). The generic part is written in 3166 lines in Java. The domain-specific parts are 422, 253, and 357 lines for HDFS, ZooKeeper and Cassandra respectively; the part for HDFS is bigger because HDFS was our first target. DESTINI's implementation cost comes from the translation mechanism (§5.1). The generic part is 506 lines. The domain-specific parts are 732 (more complete), 23, and 35 lines for HDFS, ZooKeeper, and Cassandra respectively. FATE and DESTINI interpose the target systems with AspectJ (no modification to the code base). However, it was necessary to slightly modify the systems (less than 100 lines) for two purposes: deferring background tasks while the workload is running and sending stable-state commands.

7 Conclusion and Future Work

The scale of cloud systems – in terms of both infrastructure and workload – makes failure handling an urgent challenge for system developers. To assist developers in addressing this challenge, we have presented FATE and DESTINI as a new framework for cloud recovery testing. We believe that developers need both FATE and DESTINI as a unified framework: recovery specifications require

a failure service to exercise them, and a failure service requires specifications of expected failure handling.

Beyond finding problems in existing systems, we believe such testing is also useful in helping to generate new ideas on how to build robust, recoverable systems. For example, one new approach we are currently investigating is the increased use of *pessimism* to avoid problems during recovery. For example, HDFS lease recovery would have been more robust had it not trusted aspects of the append protocol to function correctly (§6.2). Many other examples exist; only through further careful testing and analysis will the next generation of cloud systems meet their demands.

8 Acknowledgments

We thank the anonymous reviewers and Rodrigo Fonseca (our shepherd) for their tremendous feedback and comments, which have substantially improved the content and presentation of this paper. This material is based upon work supported by Computing Innovation Fellowship, the NSF under grant Nos. CCF-1016924, CCF-1017073, CCF-1018729, CCF-0747390, CNS-0722077, IIS-0713661, IIS-0803690, and IIS-0722077, the MURI program under AFOSR grant No. FA9550-08-1-0352, and gifts from Google, IBM, Microsoft, NetApp, and Yahoo!. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] HDFS JIRA. http://issues.apache.org/jira/browse/HDFS.

[3] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *EuroSys '10*.

[4] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[5] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems Export. In *OSDI '06*.

[6] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07*.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*, pages 205–218.

[8] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*.

[9] Jeffrey Dean. Underneath the Covers at Google: Current Systems and Future Directions. In *Google I/O '08*.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43.

[11] Garth Gibson. Reliability/Resilience Panel. In *HEC-FSIO '10*.

[12] Salvatore Guarnieri and Benjamin Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *Usenix Security '09*.

[13] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. FATE and DESTINI: A Framework for Cloud Recovery Testing. UC Berkeley Technical Report UCB/EECS-2010-127, September 2010.

[14] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. Towards Automatically Checking Thousands of Failures with Micro-specifications. In *HotDep '10*.

[15] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *OSDI '08*.

[16] James Hamilton. Cloud Computing Economies of Scale. In *MIX '10*.

[17] James Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA '07*.

[18] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *FAST '09*.

[19] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC '10*.

[20] Lorenzo Keller, Paul Marinescu, and George Candea. AFEX: An Automated Fault Explorer for Faster System Testing. 2008.

[21] Charles Killian, James Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*.

[22] Hairong Kuang, Konstantin Shvachko, Nicholas Sze, Sanjay Radia, and Robert Chansler. Append/Hflush/Read Design. https://issues.apache.org/jira/secure/attachment/12445209/appendDesign3.pdf.

[23] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *LADIS '09*.

[24] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI '08*.

[25] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI '07*.

[26] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing Declarative Overlays. In *SOSP '05*.

[27] Om Malik. When the Cloud Fails: T-Mobile, Microsoft Lose Sidekick Customer Data. http://gigaom.com.

[28] Paul D. Marinescu, Radu Banabic, and George Candea. An Extensible Technique for High-Precision Testing of Recovery Code. In *USENIX ATC '10*.

[29] Lucas Mearian. Facebook temporarily loses more than 10% of photos in hard drive failure. www.computerworld.com.

[30] John Oates. Bank fined 3 millions pound sterling for data loss, still not taking it seriously. www.theregister.co.uk.

[31] Xinming Ou, Sudhakar Govindavajhala, and Andrew W. Appel. MulVAL: A logic-based network security analyzer. In *Usenix Security '05*.

[32] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distrubted systems. In *NSDI '06*.

[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*.

[34] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys '06*.

[35] Hadoop Team. Fault Injection framework: How to use it, test using artificial faults, and develop new faults. http://issues.apache.org.

[36] Tom White. File Appends in HDFS. http://www.cloudera.com/blog/2009/07/file-appends-in-hdfs.

[37] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *NSDI '09*.

[38] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[39] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*.

SliceTime: A platform for scalable and accurate network emulation

Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer and Klaus Wehrle
Communication and Distributed Systems
RWTH Aachen University, Germany

Abstract

Network emulation brings together the strengths of network simulation (scalability, modeling flexibility) and real-world software prototypes (realistic analysis). Unfortunately network emulation fails if the simulation is not real-time capable, e.g., due to large scenarios or complex models. So far, this problem has generally been addressed by providing massive computing power to the simulation, which is often too costly or even infeasible.

In this paper we present *SliceTime*, our platform for scalable and accurate network emulation. It enables the precise evaluation of arbitrary networking software with event-based simulations of any complexity by relieving the network simulation from its real-time constraint. We achieve this goal by transparently matching the execution speed of virtual machines hosting the software prototypes with the network simulation. We demonstrate the applicability of *SliceTime* in a large-scale WAN scenario with 15 000 simulated nodes and show how our framework eases the analysis of software for 802.11 networks.

1 Introduction

We are still in need of adequate tools for performance investigations as well as for testing of real-world network protocol implementations and large-scale distributed systems. In this regard, the first major requirement is scalability. For example, in order to facilitate the analysis of contemporary P2P applications, such a tool needs to scale up to potentially thousands of nodes. Second, we need experimentation platforms that isolate the protocol implementation and its communication from real-world communication networks. Such strong isolation is important for the investigation of malware to prevent a potential outbreak. Isolated evaluation environments are also well suited for the analysis of software for wireless networks as unwanted disturbances on the wireless channel can be avoided.

Discrete event-based network simulation is a well-established methodology for the evaluation of network protocols. Network simulators, such as ns-3 [27] or OMNeT++ [37], facilitate the flexible analysis of arbitrary network protocols. Due to their abstract modeling approach, network simulations scale well to network sizes of up to many thousand nodes.

However, abstract simulation models focus only on the most relevant aspects of the communicating nodes. They disregard the system context of a network protocol and its run-time environment, like the influence of an operating system regarding timing, concurrent processes, and resource constraints. This fundamental concept of abstraction limits the applicability of network simulations to network performance metrics. For instance, investigations of run-time performance, resource usage, and the interoperability with other protocol implementations are difficult to obtain by solely using simulations. The strict event-based notion of network simulators also makes it generally impossible to execute arbitrary networking applications inside the simulation environment. These issues complicate performance studies that are very important for the applicability of communication systems.

Performance evaluations under real conditions are mostly carried out within network testbeds of prototype implementations. However, setting up large-scale testbeds is expensive and their maintenance is often cumbersome. Shared testbeds such PlanetLab [7], Emulab [42] and MoteLab [41] partially fill this gap. Yet their flexibility is limited due to a lack of topology controllability, shared testbed usage or insufficient scalability.

Network emulation as introduced by Fall [10] brings together the flexibility of discrete event-based of network simulation with the precision of evaluation using real-world testbeds. An event-based simulation modeling a computer network of choice is connected to real-world software prototype. Traffic from the prototype is fed to the simulation and vice versa. This way, the software prototype can be evaluated in any network that can be mod-

eled by the simulator. One fundamental issue of network emulation are the different time representations of event-based simulations and software prototypes. Event-based simulations consist of a series of discrete events with an associated event execution time. Once an event has been processed, the simulation time is advanced to the execution time of the next event. By contrast, software prototypes observe a continuously progressing wall-clock time.

Existing implementations of network emulation pin the execution of simulation events to the corresponding wall-clock time. Unfortunately, this approach is only useful if the simulation can be executed in real time. Otherwise, a simulation without sufficient computational resources will lag behind and thus be unable to deliver packets timely. Such *simulator overload* may result from complex network simulations, for example due to a high number of simulated nodes or models of high computational complexity. Simulator overload has to be prevented because deficient protocol behavior such as connection time-outs, unwanted retransmissions, or the assumption of network congestion would be the direct consequence. Moreover, even slight simulator overload may invalidate performance evaluations because the network cannot be simulated within the required timing bounds.

Speeding up the simulation to make it real-time capable is the first obvious option to deal with simulation overload. This speed-up can be achieved by supplying the simulation machine with sufficient computational resources in forms of hardware or by parallelizing the network simulation. However, we argue that this approach lacks generality because parallel processing can only scale to the degree of possible parallelism within the simulation. In addition, the amount of hardware needed for real-time execution rapidly grows with the simulation complexity, making this option inaccessible for many research institutes and individuals.

So far, network emulation has merely been an arms race between the complexity of the simulation model and the computational power of the simulation hardware. Hence, traditional approaches result in *variable hardware requirements* and *fixed execution time* (real time). By contrast, we aim at reducing the cost of precise network emulation by designing a system with *fixed hardware demands* but with *variable execution time* (real time or slower). More specifically, the main contributions of this paper are the following:

1. We thoroughly elaborate the design of *SliceTime* and its underlying concept of synchronized network emulation [39, 40] (Section 2). It eliminates the need of the network simulation to execute in real-time. This enables network emulation scenarios using simulations of any complexity. We achieve this goal by synchronizing the software prototypes with the network simulation. Using virtualization, we decouple the software prototypes' perceived progression of time from wall-clock time.
2. Our implementation of *SliceTime* (cf. Section 3) for x86 systems enables the synchronized execution of Xen-based [3] virtualized prototypes and ns-3 simulations with an accuracy down to 0.01 ms.
3. We show that *SliceTime* delivers a high degree of accuracy and transparency, both regarding timing and perceived network bandwidth (Section4). We further demonstrate in our evaluation of *SliceTime* that is run-time efficient and that the synchronization overhead stays below 10% at an accuracy of 0.5 ms.
4. We illustrate how *SliceTime* simplifies testing and performance evaluations of WiFi software for Linux by remodeling a large-scale AODV field test entirely in software. We further demonstrate the scalability of *SliceTime* by applying it to a large-scale wide-area network (WAN) scenario with 15 000 nodes (Section 5).

In Section 6 we discuss the related work before concluding this paper in Section 7.

2 *SliceTime*

We now present the design of *SliceTime*. A *SliceTime* setup incorporates three main components (cf. Figure 1): The central synchronization component (*synchronizer*), at least one virtual machine (VM) carrying a software prototype of choice, and an event-based network simulation. The synchronizer controls the execution of the network simulation and the software prototypes. In order to carry out such a synchronization, the synchronizer must interrupt the execution of the prototype or the simulation at times to achieve precise clock alignment. To enable this suspension, the software prototypes are hosted inside virtual machines for means of control.

2.1 Synchronization Component

The synchronization component centrally coordinates a *SliceTime* setup. Its task is to manage the synchronous execution of the network simulation and the attached virtual machines. It implements a synchronization algorithm to prevent potential time drifts and clock misalignments between the virtual machines and the network simulation. As choice for the synchronization algorithm, we consider solutions known from the research domain of parallel discrete event-based simulation (PDES) [11]. In this regard, two classes of synchronization are distinguished, optimistic synchronization schemes and conservative synchronization schemes.

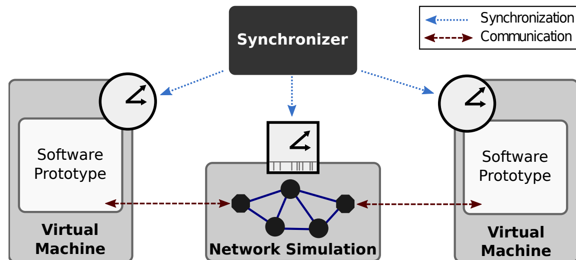


Figure 1: Conceptual Overview of *SliceTime*: By relying on entirely virtualized prototypes, we are able to synchronize the execution speed of the simulation and the prototypes. The simulation is relieved from its real-time constraint, enabling large-scale network emulation scenarios on off-the-shelf hardware.

Optimistic schemes, most notably Time Warp [18], execute the parallel simulation in a speculative fashion. In case of synchronization errors, roll-backs are used to restore a consistent and error-free global state. For the ability to roll back to a consistent state, optimistic schemes often incorporate regular snapshots of the synchronized peers. As the state of a virtual machine includes the memory allocated for the running operating system instance, check-pointing is costly at the desired level of synchronization granularity. Conservative synchronization schemes, by contrast, guarantee a parallel execution without synchronization errors, and hence, do not require a roll-back mechanism. However, most conservative schemes, such as the null-message algorithm by Chandy and Misra [6], require knowledge about the future behavior (look-ahead) of a system. While the look-ahead in event-based simulations can be determined by inspecting their event queue, predicting the future run-time behavior of a virtual machine is generally not possible. In effect, this limits the choice of a synchronization algorithm for *SliceTime* to a scheme which neither makes assumptions about the future behavior nor requires regular snapshots to be taken.

SliceTime uses a scheme similar to conservative time windows (CTW) [23] for synchronizing network simulations and VMs. In the following, we refer to this algorithm as *barrier synchronization*. Figure 2 shows the synchronization of two components, one VM and one network simulation, via the barrier synchronization algorithm. It allows every synchronized peer to run for a certain amount of time, the so-called *time slice*, after which it blocks until all other peers reach the barrier. At this point, the barrier is lifted, and a new future barrier is set up to which the execution of the synchronized components continues again. As the execution of both the network simulation and the virtual machine is always bounded by a barrier, the time drift between them is limited to the size of one time slice at all times. Consequently, the synchronization accuracy is directly given by the size of the time slice.

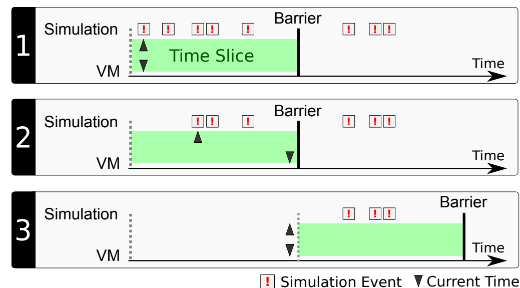


Figure 2: Different steps of the barrier algorithm used for the synchronization of one VM and one event-based simulation. The execution of the simulation and the VM is blocked until both have finished the time slice.

2.2 Virtual Machines

The virtual machines encapsulate the software prototype to be integrated with the network simulation. We consider a prototype to be an instance of any operating system (OS) that carries arbitrary network protocol implementations or applications. The virtualization of OS instances hosting software prototypes disassociates their execution from the system hardware and hence allows for obtaining full control over their run-time behavior.

Therefore, the execution of the prototype can be suspended until all synchronized components have reached the end of the time slice. This suspension avoids simulator overload by allowing the network simulator to run while the virtual machines are waiting. However, this suspension is typically detectable by the VMs, because they are relayed information from hardware time sources. Under normal circumstances, this behavior is desired to keep the clock synchronized to wall-clock time and to make sure that timers expire at the right point of time. However, since we suspend the VMs in order to synchronize their time against each other and the simulation, we must avoid this behavior. Having full control over the VM's perception of time we instead provide them with a consistent and continuous logical time. This leaves us with the possibility of transparently suspending the execution of a prototype without the implementation noticing the actual gap in real-world time.

2.3 Event-based Network Simulation

The key task of the network simulation is to model the network that connects the virtual machines. Following the terminology of Fall [10], we distinguish between an opaque and a protocol-aware network emulation mode. In the case of opaque network emulation, the simulator merely influences the propagation of network traffic,

for example by delaying or duplicating packets. This approach is prevalent in many available tools [1, 2, 5, 30]. By contrast, we focus on protocol-aware network emulation. In this case, the network simulation implements the communication protocols that are used by the VM prototypes. This enables the provision of simulated hosts that interact with the VMs.

For integrating an event-driven network simulation with a *SliceTime* setup, it needs to be interfaced to both the synchronization component (*timing control interface*) and the virtual machines (*data communication interface*). The timing control interface is tightly coupled with the event scheduler of the simulator. Recall that an event-based network simulator maintains a list of all scheduled events ordered by the time of execution. Typically, the simulation simply processes these events sequentially until the event queue is empty. In *SliceTime*, a custom scheduler checks if the next event's time of execution resides in the current time slice. If this is the case, the event is executed. If not, the event scheduler notifies the synchronization component through the timing control interface. The next event is processed after the barrier has been shifted past the execution time of the event.

The data communication interface connects the simulation and the virtual machines on the protocol level. The functional integration between the VMs and the network simulation takes place at *gateway nodes* inside the simulation, a concept adapted from [10]. These nodes can be viewed as a simulation's internal representation of the virtual machine they are connected to. Their real functionality is inside the virtual machine and their purpose is to have a communication endpoint inside the simulation at which the packet exchange with the virtual machines takes place.

For performance reasons, many network simulation frameworks use custom data structures to model a network packet, and encapsulation is mostly expressed using pointers to secondary message structures. In contrast, real systems exchange binary information, for example, Ethernet frames. When a binary packet generated by a VM arrives at the simulator, the gateway node takes care of converting it into a network simulation message. Similarly, an outgoing packet must be serialized in an adequate fashion before it leaves the simulation.

3 Implementation

We now discuss our implementation of *SliceTime* comprising three types of main components (see Figure 3): a synchronization component (*synchronizer*), the virtual machine infrastructure and a network simulation. Two different flows of communication are present in our system. The synchronizer delivers the synchronization information over the *timing control interface* using a lightweight signaling protocol. A tunnel that carries Ethernet frames from the VMs to the simulation and vice versa serves as our *data communication interface*. The VM implementation is based on the Xen hypervisor and executes multiple instances of guest domains which host an operating system and a prototype implementation. Our implementation uses the ns-3 network simulator to model the network to which the VMs are connected. For this purpose we extend the existing emulation framework of ns-3 for synchronized network emulation.

3.1 Synchronization component

The synchronizer is implemented as a user-space application. Its main purpose is to implement the timing control interface. The synchronization component assigns discrete slices of run-time to the simulation and to the virtual machines. In order to distribute the synchronized components across different physical systems, the synchronization signaling is implemented on top of UDP. In addition to the synchronization coordination, the synchronizer also manages the set of synchronized components. In particular, it allows peers to join and to leave the synchronization during run-time. This allows to run certain tasks (e.g., booting and configuring a virtual machine and the hosted software prototype) outside the the synchronized setup.

3.1.1 Timing Control Interface

One challenge is the large amount of messages that needs to be exchanged between the synchronized VMs and the simulation. For example, if the time slices are configured to a static logical duration of 0.1 ms, the synchronization component needs to issue 10 000 time slices to all attached VMs and the simulation for one second of logical time. An additional massive amount of messages is caused by the synchronized peers to signal the completion of every time slice individually to the synchronizer. Therefore, in order to maintain a good run-time efficiency, it is vital to limit the delays and the overhead caused by synchronization signaling and message parsing. For these reasons, we created a lightweight synchronization protocol based on UDP for *SliceTime*. It provides all communication primitives of the timing control interface. The assignment of time slices to all synchronized peers is carried out using UDP broadcasts, while the remaining communication, such as signaling time slice completion, takes place using unicast datagrams. Moreover, the UDP packets have a fixed structure and only carry the synchronization information in binary form. This is necessary to keep both the packet size and the parsing complexity at a very low level.

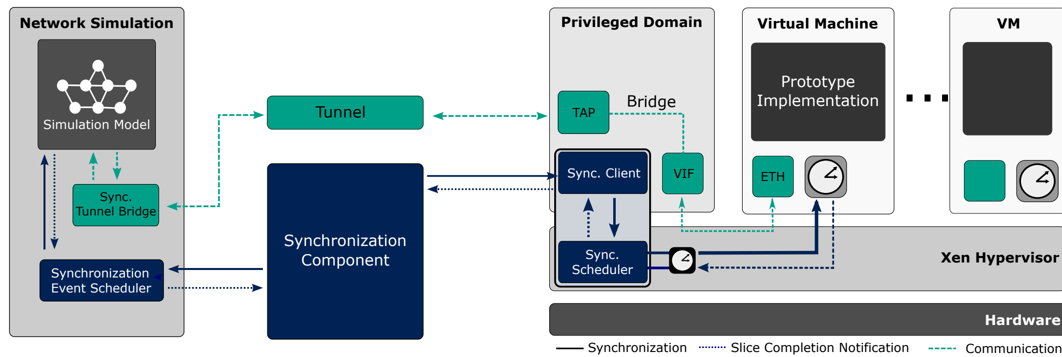


Figure 3: *SliceTime* consists of a central synchronization unit, at least one network simulation based on ns-3 and one or more Xen hypervisor systems serving as the VM infrastructure.

3.2 Virtual Machines

We use the Xen hypervisor and its scheduling mechanisms as the basis of our work. Xen is a virtual machine monitor for x86 CPUs. The hypervisor itself takes care of memory management and scheduling, while hardware access is delegated to a special privileged virtual machine (or *domain*, in Xen's parlance) running a modified Linux kernel. As the first domain that is started during booting, it is often referred to as dom0.

Xen supports two modes of operation: para-virtualization mode (PVM) and hardware virtualization mode (HVM). *SliceTime* uses Xen HVM domains for virtualizing operating systems and software prototypes. In contrast to para-virtualization, HVM Xen domains do not require the kernel of the guest system to be modified for virtualization. This allows any x86 OS, also closed source OS such as Windows, to be incorporated into a *SliceTime* set-up.

We now describe the main parts of our work: a) the data communication interface to couple virtual machines and the simulator, b) the synchronization client that interfaces with the synchronization component, and c) the changes necessary to transparently interrupt and restart the VM to align its execution speed to the run-time performance of the simulator.

3.2.1 Data Communication Interface

For the network data communication between virtual machines and simulation, it is first important to note that every virtual machine can have one or several virtual network interfaces that look like real interfaces to the virtual machine, and can be accessed inside dom0. We bridge the virtual interface in the dom0 with a tap device and redirect all Ethernet traffic from the VM to the computer running the simulation. Conversely, all Ethernet frames received from the simulation over the tunnel are fed back to the virtual machine in the same way.

3.2.2 The Xen Synchronization Client

To keep the VM in sync with the communication, the synchronization component communicates with a synchronization client on the machine running Xen. Because of the potentially high number of synchronization messages (depending on the the size of the chosen time slices), the performance of the synchronization clients is crucial to the overall performance of the system. For this reason, the client was implemented as a Linux kernel module. This is especially beneficial because Xen delegates hardware access to the privileged domain dom0. Therefore, the implementation in kernel space of the privileged domain saves half of the otherwise necessary context switches for communication and our VM implementation. Since context switches (between user space, kernel space, and, in addition here, hypervisor context) are expensive operations, halving the number of them has a very noticeable impact on the overall performance.

The client communicates with the synchronization component via UDP datagrams as described in Section 3.1.1. It then instructs Xen's scheduler via a hypercall (the domain-hypervisor equivalent of a user-kernel system call) to start the synchronized domain for the amount of time specified by the synchronizer. The client also registers an interrupt handler to a virtual interrupt, that is, an interrupt that can be raised by the hypervisor. When the synchronized domain has finished its assigned time slice, the interrupt is raised, the client's handler is executed, and it can inform the synchronizer via UDP. This interrupt-based signaling ensures a prompt processing by the involved entities.

3.2.3 Xen Extensions

The other tasks necessary for our synchronization scheme are carried out within the Xen hypervisor. To reach the goals we set forth, it is necessary to be able to precisely start and stop the VM's operation according to the assigned time slices by the synchronization

component. However, since operating systems have ways to detect the passing of time via hardware support (real time clocks, hardware timers etc.), simply stopping and restarting the VM will not lead to the desired effect. It will still be aware of the passing of time while it was stopped, and therefore, operations that depend on time information (e.g., time-outs of TCP connections) will still occur at the wrong times. Therefore, to reach transparency, it is not only necessary to be able to start and stop VMs accurately, but also to provide a consistent and steady perception of time for the VM. Hence, all time sources of the VM must be controlled and adjusted in the hypervisor.

To reach the first goal, that is, starting and stopping VMs and running them for precise number of times, we extended the sEDF (simple earliest deadline first) scheduler that is part of the Xen hypervisor. Schedulers in Xen schedule VMs in a similar fashion to an operating system's scheduler. In particular, the sEDF maintains periodical deadlines for each domain, and an amount of time the domain has to be executed up to that deadline. To manage the domains, it utilizes several queues. A run queue contains all domains that still need to run some time until their next deadline; once this constraint is fulfilled, a domain migrates to the wait queue until it reaches its deadline, at which point it rejoins the run queue with a new deadline and required execution time.

However, the synchronized domains have to be kept outside this periodical scheme, because these are only scheduled when the synchronization component issues the instruction to do so. Therefore, we introduced another queue, the sync queue, which works as a replacement of the wait queue for synchronized domains. These domains stay on that queue until they are to be scheduled again, then migrate to the run queue, and back to the sync queue afterwards. This way, synchronized domains can be kept outside the normal scheduling on non-synchronized domains. Hence non-synchronized domains may coexist with synchronized domains on the same physical machine.

One issue that originally impaired precise timing in the low microsecond range was rooted in the original implementation of the Xen scheduling subsystem. The Xen scheduler assumes itself to run instantly, not consuming any time. Therefore, a time stamp at the beginning of the execution of the scheduling loop was taken. This was considered the point of time the next scheduled domain was started. Therefore, time spent in the scheduler was attributed to the domain chosen for execution. We changed this to take a time-stamp before the context switch to the domain. This causes the time spent in the scheduler not to be attributed to any domain, therefore increasing accuracy. In addition, our modified sEDF scheduler records overall assigned run-time and adjusts itself to the small (generally sub-microsecond) inaccuracies that are inherent to Xen's timer management and lead to slightly early or late returns from the scheduled VM to the hypervisor.

To reach the second goal, that is, masking the passing of time from VMs while being stopped, different changes had to be applied to the Xen hypervisor. In fact, one of the reasons we decided to use a virtualization approach for *SliceTime* was the specific characteristic of decoupling a virtualized operating system from the hardware it, under normal circumstances, directly interfaces with. This way, we can modify the information that the OS receives from the hardware time sources, and therefore reach our goal of masking the passing of time.

To facilitate this masking, we have to amend the two main sources of time keeping: time counters and interrupt timers. Within the modified scheduler, we take timestamps whenever a domain is scheduled and unscheduled. This allows us to keep track of the total amount of time the domain was not running since the start of the synchronization. This delta value is subtracted from the counter that domains use to measure the passing of time; in the case of Xen and HVM domains, this measurement is chiefly based on the time stamp counter (TSC), a CPU register whose value increases at regular intervals. Modern CPUs with hardware virtualization support allow the virtualization of the TSC, which allows us to change its value as realized by the VM by subtracting the delta value. This way, the TSC progresses in a linear fashion, even if the domain is unscheduled for extended amounts of time.

Timers, the second source of time keeping, must also appear to act as if the domain was running continuously. To facilitate this, the same scheduler timestamps are used to keep track of the time the domain was last unscheduled. Whenever a domain is unscheduled, all timers that belong to it are stopped; in particular, all timers that belong to the virtualized hardware timers such as the RTC and APIC timers. When the domain is rescheduled again, the time delta since the last unscheduling is added to the expiry time of all timers, after which they are reactivated. This way, timers expire at the correct point of virtual time, upholding the notion of linearly progressing time.

3.3 Network Simulation

SliceTime relies on ns-3 as network simulator, as opposed to our preliminary work [39,40] in which OMNeT++ was used. In contrast to OMNeT++ and the vast majority of all event-based network simulators, ns-3 internally represents packets as bit vectors in network byte order, resembling real-world packet formats. This removes the need of explicit message translation mechanisms and simplifies the implementation of network emulation features.

The modular design of ns-3 facilitates the integration of the additional components as needed by *SliceTime*. The timing control as well as the communication interface are implemented as completely separate components whose implementation is not intermingled with existing code.

There are some similarities between the *SliceTime* simulation components and the emulation features already provided by ns-3. Both have to synchronize the event execution to an external time source. For the existing emulation implementation of ns-3 this is the wall-clock time. In the case of *SliceTime* the synchronizer acts as external time source. The so called simulator implementations in ns-3 are responsible for scheduling, unqueuing and executing events. There is one which does this in a standard manner and another one for real-time simulations (i.e., synchronized to wall clock time). Which of these is used is determined by setting a global variable in the simulation setup.

We added a third simulator implementation that connects arbitrary ns-3 simulations to the timing control interface. The simulation registers at the synchronizer before its actual run begins. Similarly, the simulation deregisters itself at the synchronizer after all events have been executed. Upon the execution of an event, our implementation checks whether its associated simulation time is in the current time slice. If this is not the case, it sends a finish message to the synchronizer and waits for the barrier being shifted. The actual communication with the synchronizer is encapsulated in a helper class which holds a socket, provides methods to establish and tear down a connection and to exchange the synchronization messages. Another modification is the provision of a method which schedules an event in the current time slice. This is needed because the regular scheduling methods only provide the time of the last executed event, which can be wrong in case of network packets arriving from outside the simulation.

The ns-3 simulator already provides two mechanisms for data communication with external systems. Both can be used with real-time simulations and synchronized emulation. The emulation net device works like any ns-3 network device, but instead of being attached to a simulated channel, it is attached to a real network device of the system running the simulation. In contrast to this the tap bridge attaches to an existing ns-3 network device and creates a virtual tap device in the host system. With both mechanisms, packets received on the host system are scheduled in the simulation and packets received in the simulation are injected into the host system.

Besides supporting these existing two ways, we added a *synchronized tunnel bridge*. It implements the data communication interface and connects the simulation to a remote endpoint. The endpoint is usually formed by a VM, however the tunnel protocol could also be used to interconnect different instances of ns-3. Again the actual communication is encapsulated in a helper class. This is not only to keep the bridge itself small, but also to reduce the number of sockets needed. In a scenario where multiple tunnel bridges are installed inside a simulation it is sufficient to have one instance of this helper class. Outgoing packets are sent through its socket to a destination specified by the bridge sending the packet. Incoming packets are dispatched by an identifier included in our tunnel protocol and then scheduled as event in the corresponding bridge to which the sender of the packet is connected. Since incoming packets are not triggered by an event inside the simulation but can occur at any time, there is a separate thread running which uses a blocking receive call on the socket. This technique has the advantage to avoid polling and is also used by the emulation net device and the tap bridge.

4 System Evaluation

We now examine the achievable accuracy of *SliceTime*. First, we look into the timing precision and the accuracy of the perceived throughput. Later on, we also measure the performance impact introduced by the synchronization process on the general run-time performance. We further investigate how it affects the perceived CPU performance on a VM. All experiments were carried out in a testbed of four Dell Optiplex 960 PCs, each equipped with a 3GHz Intel Core2 Quad CPU and 8 GB of RAM, either executing our VM implementation based on Xen or ns-3 with our synchronization extensions. The PCs were interconnected using Gigabit Ethernet. Regarding the VMs, we used Linux 2.6.18-xen for the control domain as well as the guest domains.

Most importantly, *SliceTime* needs to produce valid results for any run-time behavior of both the simulation and the VMs attached. For this purpose, we investigate two performance metrics at different levels of synchronization accuracy. The round-trip time between a simulation and a VM as well as the TCP throughput of two VMs which are communicating using TCP over a simulated network.

4.1 Timing Accuracy

In our first experiment, we captured 1 500 ICMP Echo replies (Pings) between a VM and a simulated host for different simulated link delays and time slice sizes. Figure 4 shows the measured RTT distributions for a fixed time slice size of 0.1 ms. We visualize the RTT distributions using standard box plots. The boxes are bounded by the upper and lower quartile of the corresponding RTT distribution. The box represents the middle 50% of

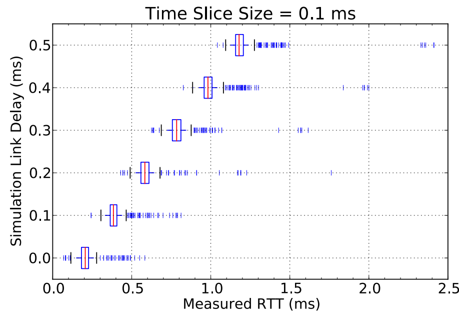


Figure 4: RTTs for different simulated link delays: the simulated delays are correctly perceived by the VM

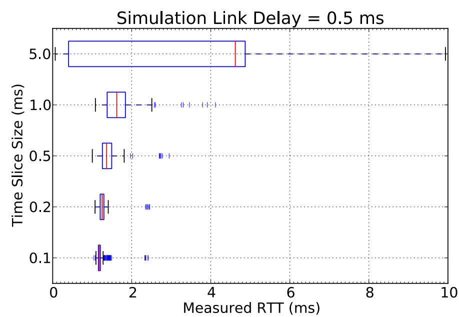


Figure 5: RTT distributions for different time slice sizes: smaller time slices lead to a higher synchronization accuracy and less variance in the measured RTTs.

the RTT measurements and its width is given by the interquartile range (IQR). The whiskers visualize the lowest and the highest RTT measured within an interval of 1.5 IQR.

If no simulation delay is present, most RTTs fall into a small range around 0.2 ms. We term this the *base delay* and it comprises time for processing and packet propagation. At all other simulation delays, the median and the RTT distributions are correctly shifted by the sum of twice the simulated link delay. For every series, few outliers are well above the expected range. We explain these deviations with the non-deterministic processing delay of ICMP frames inside the VM's protocol stack. Figure 5 displays the relation of the chosen time slice size and the resulting RTT distributions for a fixed simulated link delay of 0.5 ms and a variable time slice size.

As expected, the variation decreases for smaller time slices and converges towards the expected value of twice the simulated link delay plus the base delay. First, this result clearly demonstrates that a higher synchronization accuracy directly impacts the accuracy of the measurements themselves. Second, we see that it is important to choose the time slice size considerably smaller than the simulated link delay. Hence, the correct choice of the adequate slice size is a crucial parameter of *SliceTime*. For the simulation of many WAN scenarios (e.g., Internet services) time slices in the range between 0.1 ms and 2 ms are sufficient, as RTTs are mostly in the range of several milliseconds.

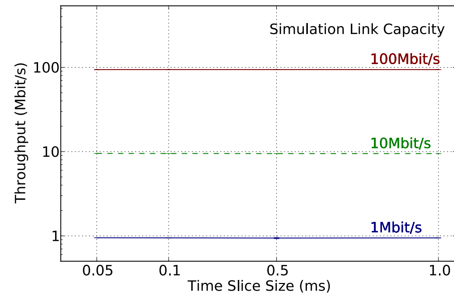


Figure 6: Network Throughput at different time slice sizes: the synchronization does not affect the throughput perceived by the VMs. The measured throughput on the VMs corresponds to the simulated link capacity.

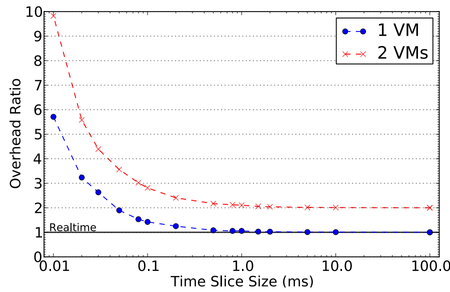
4.2 Throughput Accuracy

We now evaluate the accuracy of our implementation regarding the network throughput perceived by the VMs. For this purpose we use a small ns-3 simulation, consisting of one IP node to which two gateway nodes are attached using full-duplex CSMA/CD channels. To each of those two gateway nodes, one VM is connected. Using the netperf [19] `TCP_STREAM` benchmark, we measured the throughput between both VMs. Figure 6 shows the results for different simulated channel bandwidths and varied time slice sizes. The data points are averages over 10 netperf runs, with every run lasting 20 seconds.

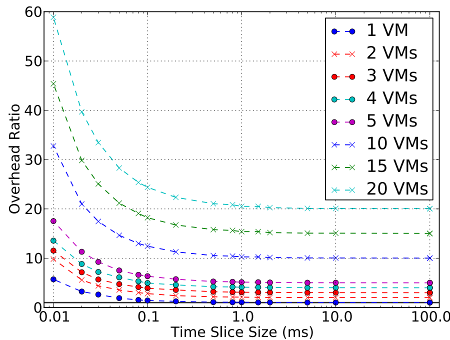
Most notably, the synchronization is transparent to the VMs in terms of perceived TCP bandwidth, as the time slice size has practically no influence on the measured TCP throughput. In addition, the throughput measured on the VMs very well reflects the simulated channel bandwidth. On average, the measured net throughput on the VMs is 5.4% lower than the simulated link capacity.

4.3 Synchronization Overhead

Because synchronized VMs are not operating in real-time, we now analyze the overhead in terms of actual run-time penalties introduced by the synchronization. We measured the real-time duration for 120 seconds of logical time issued to the VMs by the synchronizer. All VMs were executed on the same physical machine. We calculated the *overhead ratio (OR)* by dividing the consumed real-time by the logical run-time. Figure 7(a) displays the OR of one and two VMs (HVM mode) for varying time slice sizes. Up to a size of 0.5 ms, the synchronization overhead remains below 10%, which is still close to real-time behavior. For smaller slice sizes, VMs need to be



(a) 1 to 2 synchronized virtual machines



(b) 1 to 20 synchronized virtual machines

Figure 7: Overhead introduced at the VM at different synchronization levels: we observe less than 10% of run-time overhead for time slices greater than 0.5 ms. The overhead is linear in the number of VMs on one physical machine.

suspended and unpaused more frequently, and the messaging overhead increases. This leads to a higher OR.

The parallel execution of several VMs per physical machine is not the main objective of our work. Nevertheless, our implementation nevertheless facilitates such configurations. Figure 7(b) shows the OR also for a higher number of VMs. The increase of the OR is linear in the number of VMs for all time slice sizes. This is a straight consequence of our scheduling policy. Even if a system is equipped with multiple processors or cores, VMs are always executed in a pure sequential order. This is a limitation of our current implementation and we regard the parallel execution of multiple synchronized VMs as future work.

4.4 CPU Performance Transparency

One of the major reasons for the run-time efficiency of *SliceTime* is given by the fact that the VMs, once scheduled, are executed natively on the host machine instead of a full simulation of system hardware. While we have previously shown that the integration with the network simulation is accurate in terms of timing and network

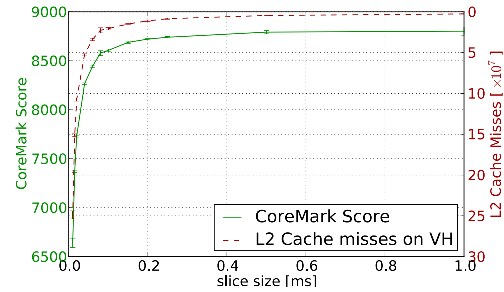


Figure 8: CoreMark CPU Benchmark score at different time slice sizes: For smaller time slices, the CPU performance of a VM decreases due to an increased amount of L2 cache misses. Please note the inverted y-axis on the right.

bandwidth, we now investigate the transparency of our VM implementation regarding the perceived CPU performance within a VM. In an ideal case, the perceived CPU performance of a VM would be invariant at different levels of synchronization accuracy.

In order to quantify the CPU performance of a VM, we executed CoreMark [34] inside the synchronized VM. CoreMark is a synthetic benchmark for CPUs and microprocessors recently made available by the Embedded Microprocessor Benchmark Consortium (EEMBC). It performs different standard operations, such as CRC calculations and matrix manipulations, and outputs a single CPU performance score. Figure 8 shows the CoreMark score for different time slice sizes. Most notably, the CPU performance is rather stable above time slices of 0.2 ms. For a time slice size of 0.1 ms, the impact of the synchronization still is less then 5%. However, for small values, the CPU performance decreases rapidly. At the highest measured accuracy level (0.01 ms), the CoreMark score drops to about 73% of the score of an unsynchronized VM on the same hardware.

We further investigated this effect using OProfile [28] and its XenoProf [25] extension. By concurrently executing OProfile in the control domain while CoreMark was running inside the VM, we were able to trace internal CPU events caused by the VM. This way, we identified an increased amount of L2 cache misses to cause the observed performance degradation. As shown in Figure 8, the number of L2 cache misses is negatively correlated to the measured CoreMark scores. For smaller time slices, the CPU needs to be switched more frequently between the execution of the VM and the control domain, thus decreasing the efficiency of L2 caching. Although this is a conceptual issue, we argue that the effect is negligible for time slices down to 0.1 ms. This means that for the vast amount of application scenarios that will use larger

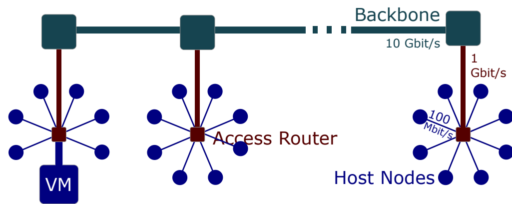


Figure 9: Simple P2P Network: the simulation consisted of one VM and 15 000 simulated nodes (60 backbone nodes with 250 host nodes each)

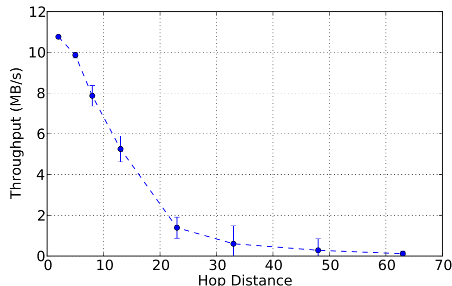


Figure 10: Throughput between VM and simulated hosts at different hopcounts

slices, this minimal performance reduction will have no negative influence on the produced results.

5 Applications

We now describe two typical use cases for *SliceTime*.

5.1 Simple P2P Network

A core motivation of our work is to enable large-scale network emulation setups on customary hardware. In order to stress our framework in this direction we first applied our framework to a large-scale WAN scenario in which 15 000 simulated nodes exchange data in a P2P-like fashion. Due to the simulation size and event load, the whole setup executes about 15 times slower than real-time. For this experiment we used just two of the four testbed machines (cf. Section 4). One machine executed the VM infrastructure and the synchronizer while the simulation was running on the other one. Figure 9 illustrates the two-tier topology we used, consisting of 60 interlinked backbone nodes, to which 250 host nodes each are attached via an access router. All host nodes act both as HTTP servers and HTTP clients, requesting a random number of 64kb data blocks from each other. To one of the access routers we connect one VM that runs a standard Linux distribution. The synchronization accuracy was set to 0.1ms. Using the standard `curl` command-line tool we measured the HTTP throughput between the virtual machine and simulated hosts at different hop distances (see Figure 10). The observation of the throughput decreasing for higher hop counts is expected and rather straightforward. However, our point here is a different one. First, we achieve valid and consistent measurements on the VM despite both the simulation and the VM operating only at a fraction of wall-clock time. Second, this simple example shows that *SliceTime* enables one to evaluate real-world networking software in a large-scale simulated context at low hardware and minor setup costs, especially if compared with equally sized physical testbeds or simulation hardware capable of executing the same simulation in real-time.

5.2 WiFi Software

SliceTime enables investigations of WiFi software for Linux in a fully isolated, deterministic and reproducible context. The 802.11 software is deployed on a set of VMs, while the network simulation models the wireless channel, the medium access control as well as potential node movement. In addition, the network simulation can optionally be used to also model other parts of the network, such as 802.11 access points, other mobile hosts or an arbitrary wide-area network connecting the 802.11 infrastructure. In the following, we briefly describe the 802.11 extensions of *SliceTime* before we use our framework to remodel a real-world field test of an AODV routing daemon for Linux.

5.2.1 *SliceTime* 802.11 extensions

To enable WiFi support in *SliceTime* we designed a second data communication interface (cf. Section 3.2.1). Figure 11 illustrates its core components and layers. On the VM a loadable kernel module forms the *SliceTime* device driver that provides a virtual WiFi interface. The device driver implements the 802.11 wireless extensions for Linux network devices. This makes the virtual WiFi interface look like a real wireless networking card. For example, commands such as `iwconfig` may be used to put the virtual WiFi device into monitor mode. The actual WiFi software may directly access this interface or rely on the Linux TCP/IP stack for its communication purposes. So-called WiFi gateway nodes represent the VMs inside the simulation. The WiFi gateway nodes perform all 802.11 MAC layer operations, for instance sending ACKs, that are normally carried out by WiFi hardware. A major benefit of this approach is that all communication events being sensitive to strict timing constraints remain in the simulation domain. Typically a relatively loose VM-simulation synchronization accuracy of 0.5ms and hence low overhead is sufficient for most *SliceTime* WiFi set-ups. By contrast, implement-

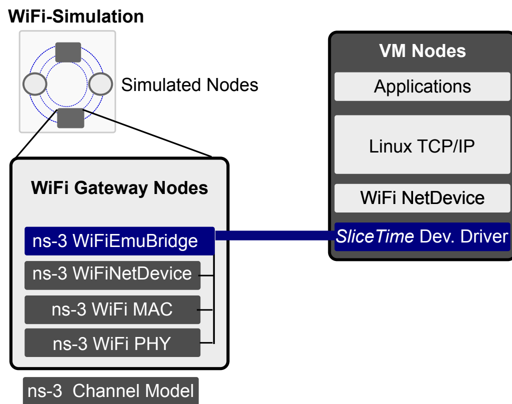


Figure 11: *SliceTime* provides a virtual network device to the VMs that integrates with ns-3 at the MAC layer. This facilitates testing arbitrary WiFi and networking software with, for example, reproducible channel conditions and node movement.

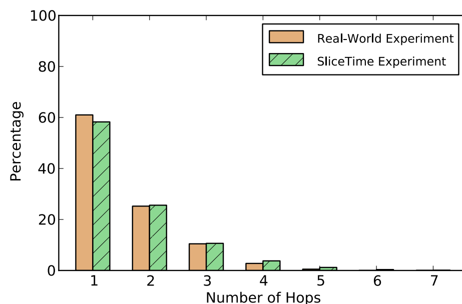


Figure 12: Real-World AODV experiment vs. remodeled *SliceTime* scenario: the hopcount distribution of received packets obtained from the scenario remodeled with *SliceTime* well matches the hopcounts measured in the real-world scenario.

ing the MAC behavior in the driver would require a synchronization accuracy lower than the 802.11 inter-frame spaces (IFS). Despite the IFS being smaller than the maximum synchronization accuracy of *SliceTime*, the high messaging overhead for such tight intervals would also render such a design impractical.

Besides implementing the data exchange between the VM device driver and the ns-3 simulation model, the WifiEmuBridge also maps configuration actions such as triggered by `iwconfig` to corresponding operations in ns-3. In addition it is able to export packet-level statistics such as RSSI values to the software running on the VM using Radiotap packet headers. A more elaborate discussion of our ns-3 WiFi emulation extensions can be found in [38].

5.2.2 AODV routing daemon study

We used *SliceTime* to remodel the AODV part of a real-world field test [13] in which different mobile ad-hoc network (MANET) routing protocol implementations were evaluated. In the original experiment volunteers on an athletic field carried around 33 laptops running an AODV daemon. The AODV routing daemon used the 802.11b ad-hoc demo mode for link layer communication. During the experiment the mobile nodes recorded both routing and traffic statistics as well as GPS traces to log the node mobility. Corresponding trace files are publicly available at the CRAWDAD repository [14]. To remodel the original experiment entirely in software using *SliceTime* we set up 33 VMs executing the AODV software bundled with the trace files from CRAWDAD. The AODV daemon was configured to use the virtual WiFi NetDevice of *SliceTime*. We implemented a corresponding simulation scenario in ns-3, which used the ns-3 log distance propagation loss model and random fading for modeling the wireless channel. In addition we extended ns-3 with a mobility model that reproduces the nodes' mobility according to the GPS traces. We only used one of our testbed machines for this experiment. It hosted all 33 VMs, the synchronizer and the ns-3 simulation. The synchronization accuracy was configured to 0.5 ms. Figure 12 compares the AODV hopcount distributions of received packets for the real-world data and the corresponding remodeled scenario. The hopcounts measured using *SliceTime* well match the observations from the real-world field test. We also determined the average packet delivery ratio (PDR) for the real-world experiment and the emulated scenario. From the CRAWDAD traces we calculated the avg. PDR to be 42.10% for the real-world AODV experiment. In our remodeled scenario the avg. PDR amounts to 46.39%.

There will always be differences between real-world measurements and observations taken with systems such as *SliceTime*. This is a direct consequence of the disparity between the real world and the environment modeled in software. The 802.11 model of ns-3, for example, is relatively sophisticated and quite accurately reproduces the behavior of the 802.11 MAC and PHY layers. However, there are many factors that are not considered by our remodeled scenario, like antenna characteristics or even a hypothetical nearby microwave that could have influenced the real-world measurements.

Nevertheless, this use case shows that *SliceTime* is well able to provide a testing environment for 802.11 software that delivers results being close to reality. Repeating real-world experiments like the one conducted by Gray [13] is costly and often challenging due to continually changing conditions, for example, regarding the wireless channel. By contrast, *SliceTime* allows one to ar-

bitrarily modify and rerun WiFi software experiments at the push of a button. *SliceTime* is also cost effective compared to the hardware costs and manpower requirements of the original experiment. While the original experiment involved around 40 volunteers and the same number of laptops, with *SliceTime* the same experiment can be conducted on one desktop PC.

6 Related Work

Early contributions [1, 17, 30, 36] in the field of network emulation focus on opaque network emulation in which physical network systems are connected to an emulation engine that models the network propagation. The model affects the packet flow, either by introducing delay, jitter, bandwidth limitations, or packet errors. Later contributions extend this methodology for the emulation of Internet paths [31] or use real-world measurements [5] for accurately reproducing the behavior of large-scale networks. Opaque network emulation is an effective method to investigate the impact of network propagation characteristics on protocol performance. However, because all communicating peers are physical systems, the analysis of large-scale scenarios (e.g., P2P and overlay networks) with many hosts is difficult.

Protocol-aware network emulation was introduced by Fall [10], proposing the combination of real network systems and discrete event-based simulations. This implementation has been improved later in terms of timing accuracy [24]. Protocol-aware emulation features also exist for other event-based network simulators [35]. All of these implementations are subject to potential simulation overload. Kiddle [21] used massive computing power in form of hardware to increase the execution speed of the simulation to circumvent this problem. While this works up to a certain point, our aim is in the opposite direction of slowing down the real system, saving on hardware expenses and setup complexity.

Erazo et al. recently proposed SVEET! [9], a hybrid TCP evaluation environment that integrates Xen-based VMs with an SSFNET [8]-based emulation engine. Although SVEET! involves a mechanism to cope with simulation overload, it differs significantly from our work. In order to match the execution speed of both the VMs and the emulation engine, SVEET! utilizes a static time dilation factor (TDF). The TDF is used to throttle down the speed of both the simulator and the VMs to the worst-case run-time performance of the emulation engine. The main drawback here is the need to correctly choose the TDF beforehand. If the chosen TDF is too large, the run-time is increased without any benefit due to under-utilization of system resources. If the chosen TDF is too small, simulation overload and time drifts can occur, leading to flawed results. In contrast, our approach does not statically throttle the execution speed of any component by a constant factor. Moreover, the conservative barrier algorithm used in our work limits the drift of all components to the duration of one time slice.

Different virtualization-based opaque network emulation approaches have been discussed over the past years. ENTRAPID [16] executes multiple instances of the FreeBSD network stack in the user space. These virtual network kernels (VNKs) are wired together and form a network emulation environment. As the VNKs are executed simultaneously and operate in wall-clock time, this limits the scalability of this approach. dONE [4] proposes the virtualization of time to address this problem. Despite this similarity *SliceTime* differs significantly from both dONE and ENTRAPID: first, neither dONE nor ENTRAPID integrate software prototypes with an event-based network simulation at all. By contrast, *SliceTime* relies on ns-3 as emulation backend. This enables the set-up of emulation scenarios that access all models and features of the network simulator. Second, in opposition to *SliceTime*, neither dONE nor ENTRAPID allow the investigation of entire network protocol stacks, as both draw the line between the emulation environment and software prototypes right at the socket layer. Diecast [15] and Time Jails [12] facilitate the setup of a network emulation testbeds solely based on virtual machines. The main advantage compared to the aforementioned systems is that they allow one to execute unmodified software and protocol stacks. Both are an attractive option for real-world experiments in which the number of nodes exceeds the quantity of physical hosts of a testbed. In addition, Diecast not only scales time, but also the performance of system components to accurately model a realistic hardware behavior profile. *SliceTime*, by contrast, follows a different goal. Instead of virtualizing time to increase the capacity of a physical testbed, we employ it for synchronizing a VM with a network simulation that forms the emulation engine. This has two advantages. First, using a network simulator as backend allows us to put concepts such as virtual node mobility into action, which is not possible with neither DieCast nor Time Jails. Second, the scalability of the simulator opens up the possibility of implementing large-scale emulation scenarios that could not be realized using VMs alone without taking up much higher hardware resources.

Emulab [42] is a well-established large network testbed allowing for the evaluation of networked software in different communication environments. Its main strength is the ability to specify network scenarios using a configuration file which Emulab maps to the testbed hardware. In order to reproduce the characteristics of networks of many kinds, Emulab also employs opaque network emulation between the testbed nodes. In direct comparison with *SliceTime*, Emulab achieves its flexibil-

ity by incorporating a huge amount of networked computers, network infrastructure as well as auxiliary components. We admire the efforts and achievements of its creators in this regard. *SliceTime* instead aims at providing a flexible and scalable network experimentation and evaluation platform with very modest hardware requirements. This is reflected in our evaluation which at most required two Desktop PCs to carry out the large-scale WAN experiment. We achieve this goal by scaling execution time and by modeling large parts of the scenario using the ns-3 simulator. On one hand the use of a simulator limits the possible degree of realism due to discrepancies between the real world and the corresponding simulation models. On the other hand relying on a simulation allows the construction of "virtual network testbeds" that are not dependant on the availability of physical hardware or real network infrastructure.

Wireless network emulation tools, such as the CMU Emulator [20], interconnect antenna connectors of standard wireless network hardware via cables. Complex hardware, mostly based on FPGAs and DSPs, is used to model the wireless channel. While this enables a quite realistic emulation, it requires complete physical hardware for each station. There is also number of pure software-based wireless network emulation tools. Most of them, such as [26,29,43], only mimic the propagation of packets on the wireless link and do not support simulated wireless stations. A few wireless network emulation systems [22,32,33] are based on event-based network simulators. They share some similarities with the WiFi extensions of *SliceTime*, but differ significantly in the way they interface the software prototypes with the 802.11 simulation. In [22,32] the 802.11 simulation model is integrated with the software at the IP layer, which prevents investigations of 802.11 software using a different routing protocol than IP. VirtualMesh [33] bridges the gap between the simulation and the WiFi software at the MAC layer, but requires the modification of all applications making use of the wireless extensions. By contrast, the 802.11 add-ons of *SliceTime* introduce a clean cut between the simulation and the prototypes at the MAC layer. This enables arbitrary WiFi software for Linux to be evaluated without any changes to the software.

7 Conclusion

In this paper we presented *SliceTime*, a platform for scalable and accurate network emulation. *SliceTime* enables the detailed analysis of protocol implementations and entire instances of operating systems inside simulated networks of arbitrary size. We achieve this goal by matching the execution speed of software prototypes encapsulated in virtual machines to the run-time performance of the event-based simulation. Our evaluation has shown that *SliceTime* is accurate as it integrates network simulations of any size with VM based prototypes regarding timing and network bandwidth in a transparent way.

SliceTime is resource efficient. We model large parts of the experiment with a simulation and match its overall execution speed to the available hardware resources. This makes it possible to conduct large-scale network emulation studies with very moderate hardware costs, especially if compared to equally sized physical testbeds.

SliceTime opens up new application areas for network emulation. In the past, only event-based simulations executing in real-time could form a basis for network emulation. This is not true for the vast majority of network simulations. For example, the computation complexity of 802.11 channel models so far hindered the use of network emulation for larger WiFi scenarios. By eliminating this burden of real-time execution, *SliceTime* allows any simulation to be used for network emulation. We have demonstrated that this extends the applicability of network emulation to large-scale WAN and 802.11 scenarios. As we believe that *SliceTime* will be useful for a number of researchers and developers, we have made the source code available to the public. It can be downloaded at `http://www.comsys.rwth-aachen.de/research/projects/slicetime`.

Acknowledgements

We express our gratitude to our shepherd Remzi Arpaci-Dusseau and our anonymous NSDI reviewers for their valuable and helpful comments. We also greatly thank Martin Lindner and Suraj Prabhakaran for conducting additional measurements and Simon Rieche and Stefan Götz for many fruitful discussions. This research was partially funded by different DFG grants and the UMIC excellence cluster, DFG EXC 89.

References

[1] Allman, M., and Ostermann, S. ONE: The Ohio Network Emulator. Technical Report TR-19972, Ohio University, 1997.

[2] Avvenuti, M., and Vecchio, A. Application-level network emulation: the emusocket toolkit. *Journal of Network and Computer Applications 29*, 4 (2006), 343–360.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Proc. SOSP'03* (Bolton Landing, NY, USA, Oct. 2003), ACM.

[4] Bergstrom, C., Varadarajan, S., and Back, G. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *Proc. PADS'06* (May 2006), pp. 19–28.

[5] Carson, M., and Santay, D. NIST Net: A Linux-based network emulation tool. *ACM Comp. Commun. Rev. 33*, 3 (2003), 111–126.

[6] CHANDY, K. M., AND MISRA, J. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering SE-5*, 5 (Sept. 1979), 440–452.

[7] CHUN, B., CULLER, D., ROSCOE, T., BAVIER, A., PETERSON, L., WAWRZONIAK, M., AND BOWMAN, M. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM Comp. Commun. Rev. 33*, 3 (2003), 3–12.

[8] COWIE, J., NICOL, D., AND OGIELSKI, A. Modeling the global internet. *Computing in Science & Engineering 1*, 1 (Jan/Feb 1999), 42–50.

[9] ERAZO, M. A., LI, Y., AND LIU, J. SVEET! a scalable virtualized evaluation environment for TCP. In *Proc. TRIDENTCOM'09* (2009), IEEE Computer Society, pp. 1–10.

[10] FALL, K. R. Network emulation in the Vint/NS simulator. In *4th IEEE Symposium on Computers and Communication* (1999).

[11] FUJIMOTO, R. M. Parallel discrete event simulation. *Commun. ACM 33*, 10 (1990), 30–53.

[12] GRAU, A., MAIER, S., HERRMANN, K., AND ROTHERMEL, K. Time Jails: A hybrid approach to scalable network emulation. In *Proc. PADS'08* (June 2008), pp. 7–14.

[13] GRAY, R. S., KOTZ, D., NEWPORT, C., DUBROVSKY, N., FISKE, A., LIU, J., MASONE, C., MCGRATH, S., AND YUAN, Y. Outdoor experimental comparison of four ad hoc routing algorithms. In *Proc. MSWiM'04* (2004).

[14] GRAY, R. S., KOTZ, D., NEWPORT, C., DUBROVSKY, N., FISKE, A., LIU, J., MASONE, C., MCGRATH, S., AND YUAN, Y. CRAWDAD data set dartmouth/outdoor (v. 2006-11-06). Downloaded from http://crawdad.cs.dartmouth.edu/dartmouth/outdoor, Nov. 2006.

[15] GUPTA, D., VISHWANATH, K. V., AND VAHDAT, A. DieCast: Testing distributed systems with an accurate scale model. In *NSDI'08* (San Francisco, CA, USA, 2008), USENIX.

[16] HUANG, X., SHARMA, R., AND KESHAV, S. The ENTRAPID protocol development environment. *INFOCOM '99* (Mar. 1999).

[17] INGHAM, D. B., AND PARRINGTON, G. D. Delayline: A wide-area network emulation tool. *Comput. Syst. 7*, 3 (1994), 313–332.

[18] JEFFERSON, D. R., AND SOWIZRAL, H. Fast concurrent simulation using the time warp mechanism. *Simulation Series, Soc. for Computer Simulation 24-26 Jan 1985 15* (1985), 63–69.

[19] JONES, R., CHOY, K., AND SHIELD, D. Netperf. [Online] Available http://www.netperf.org December 21, 2009.

[20] JUDD, G., AND STEENKISTE, P. Repeatable and realistic wireless experimentation through physical emulation. *ACM SIGCOMM Computer Communication Review 34*, 1 (2004), 63–68.

[21] KIDDLE, C. *Scalable Network Emulation*. PhD thesis, Department of Computer Science, University of Calgary, 2004.

[22] KROP, T., BREDEL, M., HOLLICK, M., AND STEINMETZ, R. JiST/MobNet: combined simulation, emulation, and real-world testbed for ad hoc networks. In *Proc. WinTECH'07* (New York, NY, USA, 2007), ACM, pp. 27–34.

[23] LUBACHEVSKY, B. D. Efficient distributed event-driven simulations of multiple-loop networks. *Comm. ACM 32* (1989), 111–123.

[24] MAHRENHOLZ, D., AND IVANOV, S. Real-time network emulation with ns-2. *8th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT)* (2004).

[25] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *Proc. VEE 2005, Chicago,USA* (2005).

[26] NOBLE, B., SATYANARAYANAN, M., NGUYEN, G., AND KATZ, R. Trace-based mobile network emulation. In *Proc. SIGCOMM'97* (1997), ACM New York, NY, USA, pp. 51–61.

[27] ns-3 Website. http://www.nsnam.org/ (accessed Oct. 2010.

[28] OProfile: a system profiler for linux. http://oprofile.sourceforge.net (accessed Oct. 2010).

[29] PUŽAR, M., AND PLAGEMANN, T. NEMAN: A network emulator for mobile ad-hoc networks. Tech. Rep. 321, Department of Informatics, University of Oslo, 3 2005.

[30] RIZZO, L. Dummynet: A simple approach to the evaluation of network protocols. *ACM Comp. Commun. Rev. 27*, 1 (1997), 31–41.

[31] SANAGA, P., DUERIG, J., RICCI, R., AND LEPREAU, J. Modeling and emulation of internet paths. In *Proceedings NSDI'09* (2009), USENIX Association, pp. 199–212.

[32] SEIPOLD, T. Emulation of radio access networks to facilitate the development of distributed applications. *JOURNAL OF COMMUNICATIONS 3*, 1 (2008), 1.

[33] STAUB, T., GANTENBEIN, R., AND BRAUN, T. VirtualMesh: an emulation framework for wireless mesh networks in OMNeT++. In *Proc. SIMUTools'09* (Brussels, Belgium, 2009), pp. 1–8.

[34] THE EMBEDDED MICROPROCESSOR BENCHMARK CONSORTIUM. CoreMark. [Online] Available http://www.coremark.org December 27, 2009.

[35] TUEXEN, M., RUENGELER, I., AND RATHGEB, E. P. Interface connecting the INET simulation framework with the real world. In *Proc. 1st International Workshop on OMNet++* (2008).

[36] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J. S., AND BECKER, D. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI'02* (2002).

[37] VARGA, A., AND HORNIG, R. An overview of the OMNeT++ simulation environment. In *SIMUTools 2008* (Marseille, France, March 2008).

[38] WEINGAERTNER, E., VOM LEHN, H., AND WEHRLE, K. Device-driver enabled wireless network emulation. In *Proc. SIMUTools 2011* (Barcelona, Spain, March 2011).

[39] WEINGÄRTNER, E., SCHMIDT, F., HEER, T., AND WEHRLE, K. Synchronized network emulation: matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev. 36*, 2 (2008), 58–63.

[40] WEINGÄRTNER, E., SCHMIDT, F., HEER, T., AND WEHRLE, K. Time accurate integration of software prototypes with event-based network simulations. In *Proc. of the Poster session at SIGMETRICS 2009* (Seattle, USA, 2009).

[41] WERNER-ALLEN, G., SWIESKOWSKI, P., AND WELSH, M. Motelab: a wireless sensor network testbed. In *Proc. IPSN'05* (Piscataway, NJ, USA, 2005), IEEE Press, p. 68.

[42] 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. *SIGOPS Oper. Syst. Rev. 36*, SI (2002), 255–270.

[43] ZHANG, Y., AND LI, W. An integrated environment for testing mobile ad-hoc networks. In *Proc. MobiHoc'02* (New York, NY, USA, 2002), ACM, pp. 104–111.

Accurate, Low-Energy Trajectory Mapping for Mobile Devices

Arvind Thiagarajan, Lenin Ravindranath, Hari Balakrishnan, Samuel Madden, Lewis Girod
MIT Computer Science and Artificial Intelligence Laboratory
{arvindt, lenin, hari, madden, girod}@csail.mit.edu

Abstract

CTrack is an energy-efficient system for *trajectory mapping* using raw position tracks obtained largely from cellular base station fingerprints. Trajectory mapping, which involves taking a sequence of raw position samples and producing the most likely path followed by the user, is an important component in many location-based services including crowd-sourced traffic monitoring, navigation and routing, and personalized trip management. Using only cellular (GSM) fingerprints instead of power-hungry GPS and WiFi radios, the marginal energy consumed for trajectory mapping is zero. This approach is non-trivial because we need to process streams of highly inaccurate GSM localization samples (average error of over 175 meters) and produce an accurate trajectory. CTrack meets this challenge using a novel two-pass Hidden Markov Model that sequences cellular GSM fingerprints directly without converting them to geographic coordinates, and fuses data from low-energy sensors available on most commodity smart-phones, including accelerometers (to detect movement) and magnetic compasses (to detect turns). We have implemented CTrack on the Android platform, and evaluated it on 126 hours (1,074 miles) of real driving traces in an urban environment. We find that CTrack can retrieve over 75% of a user's drive accurately in the median. An important by-product of CTrack is that even devices with no GPS or WiFi (constituting a significant fraction of today's phones) can contribute and benefit from accurate position data.

1 Introduction

With the proliferation of sensor-equipped smartphones, the decades-long promise of location-based mobile services and mobile sensing applications is finally becoming real. Many location-based applications periodically probe the device's position sensor to obtain a stream of position samples, and then process this stream to obtain a trajectory. Examples include crowd-sourced traffic and navigation applications [15, 33], personalized trip management applications [28, 15], fleet management applications [21], and mobile object/asset tracking [11, 34, 7, 19, 25]. The fundamental problem in these applications is *trajectory mapping*, where the goal is to produce the most likely trajectory—a sequence of map segments—traversed by the mobile device.

If each device could always use a GPS sensor, this problem is straightforward because the majority of the position samples would usually be accurate to within a small number of meters. For applications that require positions to be monitored continuously, however, GPS has some significant practical limitations. First, GPS chipsets on today's mobile devices consume a non-trivial amount of energy, causing a significant reduction in battery life (§2). Second, in many embedded tracking applications, objects are packaged deep inside vehicles and do not have a clear line-of-sight to GPS satellites e.g., anti-theft systems on vehicles (often hidden under layers of metal), systems that track couriered packages [11] and systems like TrashTrack [34] for tracking waste and recycled materials. Most of these tracking applications also face energy and cost constraints. Third, antenna limitations on commodity mobile devices cause poor GPS performance in "urban canyons" and near high-rise buildings. Finally, a large number of phones today simply do not have GPS on them—85% of phones shipped in 2009, and projected to be over 50% for the next five years [6]. The users of these devices, a disproportionate number of whom are in developing regions, are largely being left out of the many new location-based applications.

This paper describes the design, implementation, and experimental evaluation of *CTrack*, a system for mapping the trajectory of mobile devices without using GPS. The noteworthy aspect of CTrack is that it uses much less energy than current approaches, which use GPS, WiFi localization [32, 8], or a combination of the two. CTrack processes a stream of raw, highly inaccurate position samples from mobile devices obtained by fingerprinting cellular GSM base stations, and matches them to segments on a known map in a way that achieves high accuracy. The marginal energy cost of gathering a fingerprint (a list of nearby GSM towers and their signal strengths) is zero on mobile phones because the cellular radio is usually always on. CTrack optionally augments GSM fingerprints with data from one or more of a phone's accelerometer, compass, and gyro, all of which consume tiny amounts of energy, using these sensor hints

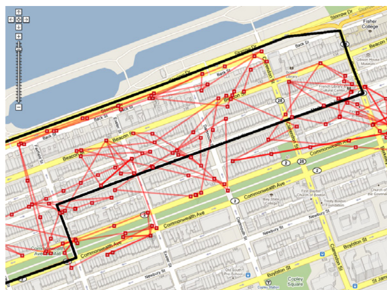


Figure 1: GSM Localization Errors. Raw location samples are in red and the true driving path is in black.

to identify the kind of movement and improve the accuracy of trajectory mapping.

GSM localization using, for example, the Placelab [8] approach, leads to errors of 100–200 meters in dense urban areas, and as much as 1 km in some areas. Such errors are too large for many applications, which require results with sufficient accuracy to pinpoint a specific road segment or route driven by a user. Figure 1 illustrates the problem with existing GSM localization. The red points are raw locations obtained from our implementation of cellular positioning as used in Placelab [8]. The actual roads traversed (ground truth) are shown in black. Directly reporting raw positions or matching locations to the nearest segments in the road map would result in unacceptably low accuracy for the applications mentioned at the beginning of this section.

CTrack makes it possible to use GSM fingerprints for accurate trajectory mapping using two novel ideas. Like previous approaches e.g. VTrack [32], CTrack matches a sequence of GSM tower observations, rather than a single point at a time, using constraints on the transitions a moving vehicle can make between locations. However, unlike VTrack, which first converts radio fingerprints to (lat, lon) coordinates, CTrack matches cellular fingerprints *directly* to a map *without* first converting them into (lat, lon) coordinates, an insight critical to achieving high accuracy. Instead, CTrack uses a two-pass algorithm. The first pass is a Hidden Markov Model (HMM) that divides space into grid cells, and determines the most likely sequence of traversed grid cells. The second pass uses a different HMM to match the traversed grid cell sequence to road segments.

The second idea in CTrack is to (optionally) *fuse* information from two low-energy phone sensors: the *accelerometer* and a *compass* or *gyroscope*. *CTrack* uses the compass/gyro to detect if the driving path took a turn, and the accelerometer to determine if the user is stopped or moving. These sensor hints can correct some common systematic errors that arise in GSM localization.

We implemented *CTrack* on the Android smartphone platform, and evaluated it on nearly 125 hours of real drives (1,074 total miles) from 20 Android phones in the Boston area. We find that:

1. *CTrack* is good at identifying the sequence of road segments driven by a user, achieving 75% precision and 80% recall accuracy. This is significantly better than state-of-the-art cellular fingerprinting approaches [8] applied to the same data, reducing the error of trajectory matches by a factor of 2.5×.
2. Although *CTrack* identifies the exact segment of travel incorrectly 25% of the time, trajectories produced by *CTrack* are on average only 45 meters away from the true trajectory. This implies that our system is useful for applications like route visualization. In this respect, CTrack is 3.5× better than map-matching raw cellular fingerprints, which results in 156 meters median error.
3. *CTrack* has a significantly better energy-accuracy trade-off than sub-sampling GPS data to save energy, reducing energy cost by a factor of 2.5× for the same level of accuracy.

2 Why Cellular?

One of the key motivations for *CTrack* is that it uses substantially less energy than GPS. This is to be expected from a theoretical standpoint because of the difference in effective radiated power (ERP) for the two systems. GPS satellites fly in an orbit 11,000 miles above the earth, with a transmission power of 50 W, resulting in 2×10^{-11} mW/m^2 at the receiver; in contrast, typical cellular systems register an ERP of up to 10 mW/m^2 [14]. This difference of 117 dB translates directly into energy consumption at the receiver, as the difference must be compensated by additional processing gain and amplification. The ERP difference also explains why GPS signals cannot be acquired without relatively unobstructed line-of-sight to orbiting satellites, and why they are more sensitive to weather conditions than GSM signals.

2.1 Energy Measurements

We performed a simple experiment to quantify the energy consumption of each of the sensors of interest — GPS, WiFi, GSM, the compass and the accelerometer on an Android G1 phone. For each sensor, we wrote an Android application to continuously sample the sensor at some given frequency, as well as continuously query the battery level indicator. We charged the phone to 100%, configured the screen to turn off automatically when idle (the default), and started the application. We used the Android telephony API to retrieve nearby cell towers and their associated signal strength values.

Figure 2 shows the reported battery life as a function of time for four configurations: GPS sampled every second, GPS sub-sampled every two minutes, WiFi scanned every second, and the configuration used by *CTrack* — scanning GSM cell towers every second, and the com-

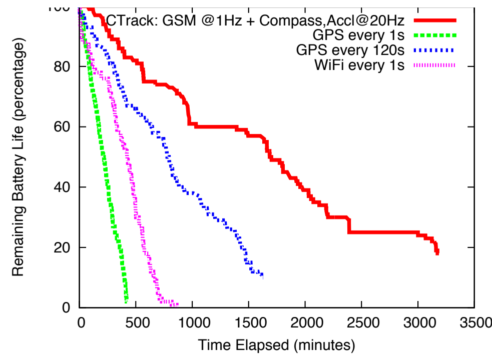


Figure 2: Energy Consumption: GPS vs WiFi vs CTrack on an Android Phone.

pass and accelerometer at 20 Hz. CTrack results in a saving of approximately 10× in battery life compared to GPS every second over 6× compared to WiFi every second. Also, although sub-sampling GPS ever 2 minutes saves energy over continuously sampling it, we show later that sub-sampling also hurts accuracy. The battery drain curves look irregular because the G1 phone estimates remaining battery life poorly – the same experiment on a Nexus One (a later model) showed a similar trend, but looked like a straight line for all sensors.

2.2 Other Energy Studies and Discussion

The numbers above are consistent many previous studies conducted on a range of phones. For example, we found [32, 31] that continuously sampling GPS on iPhone 3G and 4 resulted in 3–10 hours total battery life (iPhone 3G has lower battery life, and screen brightness varied in the different papers, resulting in different run times even without GPS). Leaving the phone on (with screen on) resulted in 10–18 hours of lifetime (this would be higher if we could turn the phone's screen off, but at the time, non-jailbroken iPhones did not support background applications.)

In [23], the authors showed that Nokia N95 phones use about 370 mW of power when GPS is left on, versus 60 mW when idling, and that continuous (once a second) GPS sampling results in 9 hours of total battery life. Several other papers [36, 16, 5, 9, 13] suggest similar numbers for N95 phones (battery life in the 7–11 hour range) with regular GPS sampling. On a more recent AT&T Tilt phone [18], the authors found that continuous GPS sampling used 400 mW, a single GPS fix costs 1.4-5.7 J of energy (depending on whether previous seen satellite information is cached or not) and a WiFi scan consumed about 0.55 J of energy.

The energy cost of GPS is rooted in the need for processing gain to acquire the positioning signals. As signal quality degrades due to obstructions or weather conditions, the energy cost of recovering the signal increases. In contrast, because phones continuously track cell towers as a part of normal operation, the marginal energy cost of CTrack is driven by CPU load. Processing a cell tower signature might require at most 100,000 instructions, which costs 5 nJ on a current generation 1 GHz Qualcomm Snapdragon processor.

In embedded (non-phone) applications that don't need the radio on, it is possible to track only the signal quality and cell ID portions of the GSM protocol. This requires observing only the BCH slots of the GSM beacon channel, which are 4.6 ms long and are transmitted once per each 1.8 second cycle. A 10% GSM receiver duty cycle should be adequate to track the strongest towers. Assuming a GSM receiver uses 17 mA at 100% duty cycle, this represents an additional power consumption of 5 mW (1.7 mA @ 2.7 V)amortized cost assuming 17 mA cost for receiver circuitry [1, 30].

Accelerometers and compasses (magnetometers) also have low overhead—for example ADXL 330 accelerometers use about 0.6 mW when continuously sampling, and at 10 Hz can be idle about 90% of the time, suggesting a power overhead of around .06 mW for sampling the accelerometer [2]. The MicroMag3 compass uses about 1.5 mW in continuous sampling, suggesting a power consumption of .15 mW or less at 10 Hz [24].

In summary, the power consumption of cellular scanning plus sensors on phones is less than 5 mW, and the power consumption of sensors alone if cellular is free—as is typical—is less than 1 mW, low enough that it does not reduce the phone's overall lifetime even when in standby mode, when it consumes 20–30 mW of power. In contrast, the best case for GPS is 75 mW in tracking mode when a fix is already acquired, but in practice is closer to 400 mW when including the energy to periodically re-acquire fixes, and is similar for WiFi scans every second or two. The power differential is thus significant.

2.3 Embedded Low-Power Applications

CTrack can also be applied outside the smartphone context to embedded low-power tagging applications. For these applications, minimizing cost and battery requirements is essential. These applications benefit from using GSM in place of GPS because of increased flexibility of antenna placement for cellular systems, and resilience to obstructed environments.

One such application is cold-chain management where the focus is on monitoring the temperature of a package during its shipping. A low-power passive cellular receiver can be used to record cellular fingerprints during transport. Upon arrival, CTrack can be run on the fingerprints to compute the shipment's trajectory and map temperature readings on to it.

Another embedded application of CTrack is TrashTrack [34, 7], where items of trash were tagged with active tags that traced the items through the path along

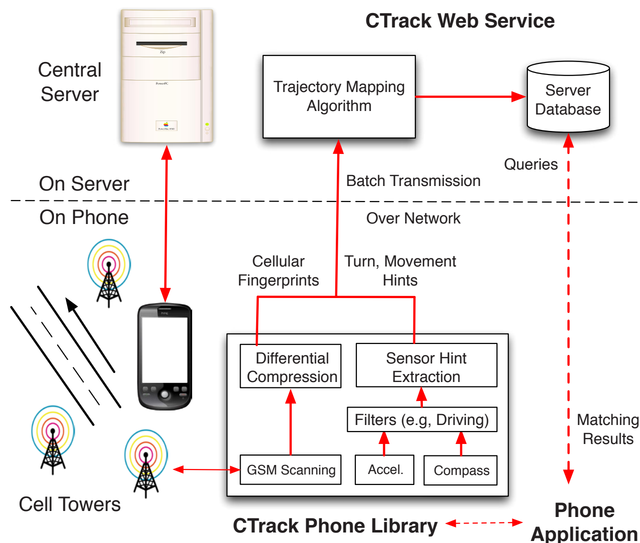


Figure 3: CTrack System Architecture.

the "disposal chain". Because the tag will eventually be destroyed, this system needs cellular communication capabilities; using the same technology for trajectory mapping consumes lower power, has lower cost, and is more robust than adding a GPS receiver to the tag.

3 System Overview

We now describe the design of CTrack. Figure 3 shows the system architecture. It consists of two software components, the *CTrack Phone Library*, and the *CTrack Web Service*. The library collects, filters, and scans for GSM and sensor data on the phones, and transmits it via any available wireless network (3G, WiFi, etc.) to the web service, which runs the trajectory mapping algorithm on batches of sensor data to produce map-matched trajectories. The mapping algorithm runs on the server to avoid storing complete copies of map data on the mobile device, and to provide a centralized database to which phone or web applications can connect to view and analyze matched tracks (e.g., for visualizing road traffic or the path taken by a package or vehicle).

Phone Library: The phone library collects a list of GSM towers and optionally, if accelerometer, compass, or gyro are available on the phone, current sensor hints. These sensor hints are binary values indicating if the phone is moving and/or turning; Section 5 describes how we extract sensor hints. The phone library also filters accelerometer data to detect if the user is stationary or walking (as in [27, 31]), for applications that want data only from moving vehicles. The library may also be configured to periodically collect GPS data for use in the training phase of our algorithm from users who wish to contribute.

Our implementation collects about 120 bytes/second of raw ASCII data on average. This quantity varies because the number of cell towers visible varies with location. We use simple gzip compression, which on our test drives resulted in just 11 bytes/second of data to be delivered. We batch this data and upload a batch every t seconds. At 11 bytes/sec, with even small batches, using a 3G uplink with an upload speed of 30 kBytes/s (typical of most current 3G networks in the US) results in very low 3G radio duty cycles—for example, setting t to 60 seconds results in the radio being awake only 0.03% of the time, which consumes a negligible amount of additional power. Once-per-minute ($t = 60$) reporting is sufficient for most applications we are concerned with, including traffic reporting, package tracking, and vehicular theft detection.

We chose not to run trajectory matching on the phone because it results in a negligible space savings, while consuming extra CPU overhead and energy. For low data rates, the primary determinant of 3G or WiFi transmission energy is the transmitter duty cycle [4], making batch reports a good idea. However, we do extract sensor hints on the phone because the algorithms for hint extraction are simple and add negligible CPU overhead, while significantly reducing data rate. The raw data rate from sampling the accelerometer/compass without compression or hint extraction is about 1.3 MBytes/hour, which means that an application collecting this data from a user's phone for two hours a day could easily rack up a substantial bandwidth bill without on-phone filtering.

CTrack Web Service: The web service receives GSM fingerprints and converts them into map-matched trajectories using the trajectory mapping algorithm. These matched trajectories are written into a database. Optionally, the user's current segment can be sent directly back to the phone. A detailed description of the trajectory mapping algorithm is given in the next section.

4 Trajectory Mapping Algorithm

CTrack's algorithm for map-matching a sequence of GSM cell tower observations ("cellular fingerprints") differs from previous approaches in two key ways. First, we do not convert cellular fingerprints into (lat, lon) coordinates before matching them to segments. We find that reducing a fingerprint to a single geographic location loses a lot of information because a given cellular fingerprint is often seen from multiple locations quite far apart. This situation is unlike the WiFi map-matching in VTrack [32], where this spread is small, and the approach of converting to centroids worked well. Second, CTrack optionally fuses sensor hints from the accelerometer and the compass to improve matching accuracy. We show that turn hints can help remove spurious turns and kinks from GSM-mapped trajectories, and movement hints can

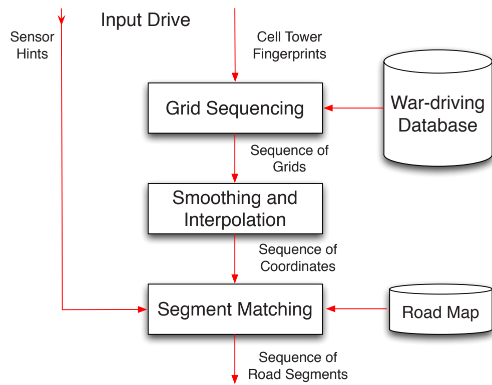


Figure 4: Trajectory Mapping Algorithm.

help remove loops, a common problem with GSM localization when a vehicle is stationary.

4.1 Algorithm Outline

The goal of the algorithm is to associate a sequence of cellular fingerprints to a sequence of road segments on a known map. Our algorithm takes as input:

1. A series of GSM fingerprints from the phone, one per second in our implementation. In our paper, the term *GSM fingerprint* refers to a set of observed IDs of cell towers and their associated received signal strength (RSSI) values. In our implementation, the Android OS gives us the cell ID and the RSSI of up to 6 neighboring towers in addition to the associated cell tower. Each RSSI value is an integer on a scale from 0 to 31 (higher means higher signal-to-noise ratio).

2. If available, time series signals from accelerometer, compass, and gyroscope sampled at 20 Hz or higher. These are converted to "sensor hints" using on-phone processing as explained below.

3. A known map database that contains the geography of all road segments in the area of interest, such as OpenStreetMaps [22], NAVTEQ, or TeleAtlas.

The output is the likely sequence of road segments traversed, one for each time instant in the input.

Figure 4 shows the components of the algorithm. *Training* builds a training database, which maps ground truth locations from GPS to observed cell towers and their RSSI values. *Grid Sequencing* uses a Hidden Markov Model (HMM) to determine a sequence of spatial grid cells corresponding to an input sequence of GSM fingerprints. The output of grid sequencing is smoothed, interpolated, and fed to *Segment Matching*, which matches grid cells to a road map using a different HMM.

Figure 5 illustrates our algorithm by example. The input "raw points" in Figure 5(a) are shown only to illustrate the extent of noise in the input data. They are not actually used by CTrack. They are computed by using the Placelab fingerprinting algorithm [8], where a cell tower fingerprint is assigned a location equal to the centroid of the closest k fingerprints in the training database (we used $k = 4$).

Next, we describe each stage of the algorithm.

4.2 Training

We divide the geographic area of interest into uniform square grid cells of fixed size g_s. We associate with each cell an ordered pair of positive integers (x, y), where $(0, 0)$ represents the south-west corner of the area of interest. We use $g_s = 125$ meters, chosen to balance running time, which increases with smaller grid size, against accuracy.

We train CTrack for the area of interest using software on mobile phones that logs a timestamped sequence of ground truth GPS locations and associated cell tower fingerprints. For each grid G in the road map, our training database stores F_G, the set of distinct fingerprints seen from G. Training can be done out-of-band using an approach similar to the Skyhook [29] fleet. Once the training database is built, it can be used to map-match or track any drive, and needs to be updated relatively infrequently. We can also collect new training data in-band from consenting participating phones that use the CTrack web service whenever the user has enabled GPS.

4.3 Grid Sequencing

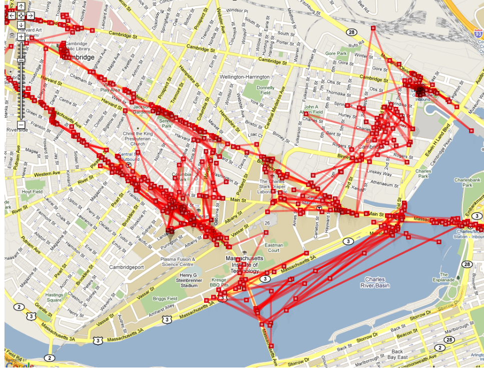
Grid sequencing uses a Hidden Markov Model (HMM) to determine the sequence of grid cells corresponding to a timestamped sequence of cellular fingerprints. An HMM is a discrete-time Markov process with a set of *hidden states* and *observables*. Each state *emits* an observable, whose likelihood is given by an *emission score*. An HMM also permits transitions among its hidden states at each time step. These transitions are governed by a different set of likelihoods called *transition scores*.

In our (first) HMM, the hidden states are grid cells and the observables are GSM fingerprints. The emission score, $E(G, F)$ captures the likelihood of observing fingerprint F in cell G. The transition score, $T(G_1, G_2)$, captures the likelihood of transitioning from cell G_1 to G_2 in a single time step.

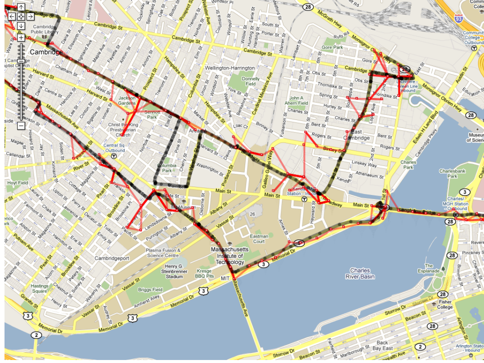
We first process the input GSM fingerprints using the *windowing* technique described below. We then use Viterbi decoding [35] to find the maximum likelihood sequence of grid cells corresponding to the windowed version of the input sequence. The maximum likelihood sequence is defined to be the sequence that maximizes the product of emission and transition scores.

We now describe the four parts of this HMM: windowing, hidden states, emission score, and transition score.

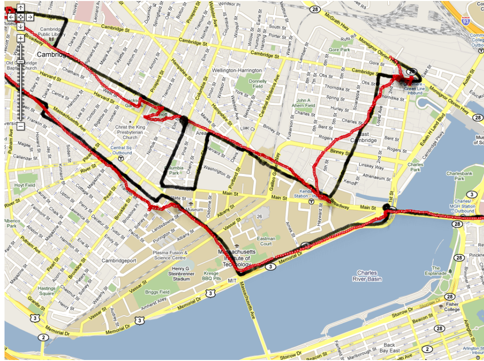
Windowing. Because it is common for a single cell tower scan to miss some of the towers near the current location, we group the fingerprints into *windows* rather than use the raw fingerprints captured once per second.



(a) Raw points before sequencing



(b) After Grid Sequencing



(c) After smoothing

(d) Final map-matched output

Figure 5: CTrack map-matching pipeline. Black lines are ground truth and red points/lines are obtained from cellular fingerprints.

We aggregate the fingerprints seen over W_{scan} seconds of scanning. We chose $W_{scan} = 5$ seconds empirically: the phone typically sees all nearby cell towers within 3 scans, which takes about 5 seconds. In our evaluation, we show that windowing improves accuracy (Table 1).

Hidden States. The hidden states of our HMM are grid cells. Given an observed fingerprint F, a grid cell G is a candidate hidden state for F if there is at least one training fingerprint in G that has at least one cell tower in common with F. Note that we might sometimes omit a valid possible hidden state G if the training data for G is sparse. To overcome this problem, we use a simple wireless *propagation model* to predict the set of cell towers seen from cells that contain no training data. The model computes the centroid and diameter of the set of all geographic locations from which each cell tower is seen in the training data. The model draws a "virtual circle" with this center and diameter and assumes that all cells in the circle see the tower in question.

Emission Score. Our emission score $E(F,G)$ is intended to be proportional to the likelihood that a fingerprint F is observed from grid cell G. A larger emission score means that a cell is a more likely match for the observed fingerprint. Our emission score uses the following heuristic. We find F_c, the closest fingerprint to F seen in training data for G. "Closest" is defined to be the value of F_c that maximizes a pairwise emission score $E_P(F,F_c)$. Our pairwise score is inspired by RADAR [3]. It captures both the number of matching cell IDs, M, between two fingerprints, and the Euclidean distance d_R in between the signal strength vectors of the matching towers:

$$E_P(F_1,F_2) = M\lambda_{match} + (d_R^{max} - d_R(F_1,F_2)) \qquad (1)$$

where λ_{match} is a weighting parameter and $d_R^{max} = 32$ is the maximum possible RSSI distance. A higher number of matching towers, and a lower value of d_R, both correspond to a higher emission score. The maximum value of the pairwise emission score is normalized (described below) and assigned as the emission score for F.

As an example, consider the fingerprints {(ID=1, RSSI=3), (ID=2, RSSI=5)} and {(ID=1, RSSI=6), (ID=2, RSSI=4), (ID=3, RSSI=10)}. The distance between them would be $2\lambda_{match} + (32 - \frac{\sqrt{(3-6)^2+(5-4)^2}}{2})$. The weighting parameter affects how much weight is given to tower matches versus signal-strength matches: we chose $\lambda_{match} = 3$.

We normalize all our emission scores to the range $(0,1)$ to ensure that they are in the same range as transition scores, which we discuss next.

Transition Score. Our transition score is given by:

$$T(G_1,G_2) = \begin{cases} \frac{1}{d(G_1,G_2)} & , G_1 \neq G_2 \\ 1 & , G_1 = G_2 \end{cases}$$

where $d(G_1, G_2)$ is the Manhattan distance between grid cells G_1 and G_2 represented as ordered pairs (x_1, y_1) and (x_2, y_2). The transition score is based on the intuition that, between successive time instants, the user either stayed in the same cell or moved to an adjacent cell. It is unlikely that jumps between non-adjacent cells occur, but we permit them with a small probability to handle gaps in input data.

Figure 5(b) shows the output of the grid sequencing step for our running example. As we can see, sequencing removes a significant amount of noise from the input data. In our evaluation, we demonstrate that the sequencing step is critical (Figure 11).

4.4 Smoothing and Interpolation

This component takes a grid sequence as input and converts it into a sequence of (lat, lon) coordinates that are then processed by the *Segment Matching* stage.

Smoothing filter. For each grid in the sequence, we calculate the centroid of the training points seen from the grid. The centroid has the following advantage: if there is only one road segment in a grid (a frequent occurrence) and the training points lie on it, so will the centroid. Typically, centroids from grid sequencing have high frequency noise in the form of back-and-forth transitions between grids (Figure 5(b)). Hence, we apply a smoothing low-pass filter with a sliding window of size W_{smooth} to the centroids calculated as described above. The filter computes and returns the centroid of centroids in each window. This filter helps us to accurately determine the overall direction of movement and filter out the high frequency noise. We chose the filter window size, $W_{smooth} = 10$, empirically.

Interpolation. Earlier, we windowed the input trace and grouped cellular scans over a longer window of W_{scan} seconds. As a result, the smoothing filter produces only one point every W_{scan} seconds. We linearly interpolate these points to obtain points sampled at a 1-second interval, and pass them as input to the *Segment Matching* step described in §4.5.

The reason for interpolation is that segment matching produces a continuous trajectory where each segment is mapped to at least one input point. The minimum frequency of input to the segment matcher is one that ensures that even the smallest segment has at least one point. The smallest segment in the OpenStreetMaps and NAVTEQ maps is roughly 30 meters; so assuming a maximum speed of 65 MPH = 105 km/h = 29 m/s, we need about once-a-second sampling or higher to ensure this condition. Higher speeds than that generally occur on freeways where segments are usually longer than 30 meters.

Figure 5(c) shows the example drive after smoothing and interpolation. This output is free of back-and-forth transitions and correctly fixes the direction of travel at each time instant. Our evaluation quantifies the benefit of smoothing (Table 1).

4.5 Segment Matching

Segment Matching maps sequenced, smoothed grids from the previous stages to road segments on a map. It takes as input the sequence of points from the *Smoothing and Interpolation* phase, and turn and movement hints from the phone, to determine the most likely sequence of segments traversed. We describe how movement and turn hints are extracted in Section 5.

For segment matching, we use a version of the VTrack algorithm [32] augmented to process sensor hints. This step also uses an HMM. In this case, the states are the set of possible triplets $\{S, H_M, H_T\}$, where S is a road segment, $H_M \in \{0,1\}$ is the current movement hint, and $H_T \in \{0,1\}$ is the current turn hint.

The emission score of a point (lat, lon, H_M, H_T) from a state (S, H'_M, H'_T) is zero if $H_M \neq H'_M$ or $H_T \neq H'_T$. Otherwise, we make it Gaussian, with the form e^{-D^2}, where D is the distance of (lat, lon) from road segment S.

The transition score between two triplets $\{S^1, H^1_M, H^1_T\}$ and $\{S^2, H^2_M, H^2_T\}$ is defined as follows. It is 0 if segments S^1 and S^2 are not adjacent, disallowing a transition between them. This restriction ensures that the output of matching is a continuous trajectory. For all other cases, the base transition score is 1. We multiply this score with a *movement penalty*, $\lambda_{movement}(0 < \lambda_{movement} < 1)$, if $H^1_M = H^2_M = 0$ and $S_1 \neq S_2$, to penalize transitions to a different road when the device is not moving. We also multiply with a turn penalty, $\lambda_{turn}(0 < \lambda_{turn} < 1)$ if the transition represents a turn, but the sensor hints report no turn. We used $\lambda_{movement} = 0.1$ and $\lambda_{turn} = 0.1$. Our algorithm is not very sensitive to these values, since the penalties are multiplied together and a small enough value suffices to correct incorrect turn/movement patterns.

Similar to VTrack, the HMM also includes a *speed constraint* that disallows transitions out of a segment if sufficient time has not been spent on that segment. The maximum permitted speed can be calibrated depending on whether we are tracking a user on foot or in a vehicle.

The output of the segment matching stage is a set of segments, one per fingerprint in the interpolated trace (which, on average, is the same periodicity as the original input). The output for the running example is shown in Figure 5(d).

When running online as part of the CTrack web service, the segment matcher takes turn hints and sequenced grids as input in each iteration and returns the current segment to an application querying the web service.

Running time. The run-time complexity of the entire algorithm, including all stages, is $O(mn)$, where m is

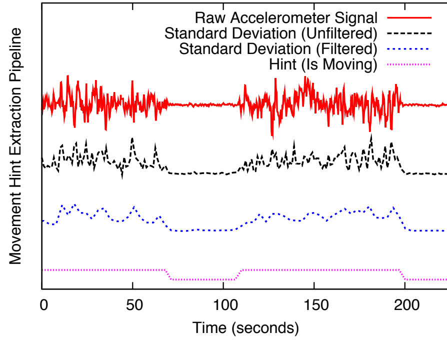


Figure 6: Movement hint extraction from accelerometer.

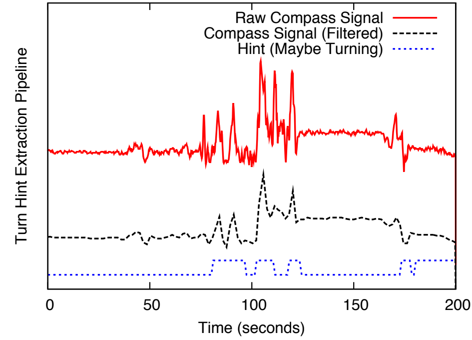


Figure 7: Turn hint extraction from compass.

the number of input fingerprints and n is the number of search states (the larger of the number of grid cells and road segments on the map). Our Java implementation on a MacBook Pro with 2.33 GHz CPU and 3 GB RAM map-matched an hour-long trace in approximately two minutes, approximately 30 times faster than real time. It is straightforward to reduce the run time by more aggressively pruning the search space, but we have not found the need to do so yet.

5 Sensor Hint Extraction

CTrack includes a sensor hint extraction layer that processes raw phone accelerometer readings to infer information about whether the phone being tracked is moving or not, and processes orientation sensor readings from a compass or a gyroscope to heuristically infer vehicular turns. These hints are transmitted along with the GSM fingerprint to the server for map matching.

Anomaly detection. Anomaly detection filters out periods when the user is lifting the phone, speaking on the phone, texting, waving the phone about, or otherwise using the phone. We want to use accelerometer and compass/gyro data only in periods where we have high confidence that the phone is more or less at rest relative to the moving object in which it is located (e.g., on a flat surface or in a user's pocket). We found empirically that when driving with the phone at rest in a vehicle or in a pocket, the raw accelerometer magnitude tends to be smaller than 14 ms^{-2}. Hence, we look for spikes in the raw accelerometer magnitude that exceed a threshold of 14 ms^{-2}. Whenever we encounter such a spike, we ignore all accelerometer and compass data in the map-matching algorithm until the phone comes back to a state of rest (this can be detected using standard deviation of acceleration, as explained below). On more recent phones such as the iPhone 4, the in-built gyroscope gives the exact orientation of the phone which can be directly read to determine if the phone is on a flat surface/in a user's pocket.

Having filtered out anomalous periods, the hint extraction processes stable periods to extract movement and turn hints, as explained below.

Movement Hints. Our algorithm uses accelerometer data sampled at 20 Hz. We extract a simple "static" or "moving" (1-bit hint) rather than integrating the accelerometer data to compute velocities or processing it in a more complex way, because accelerometer data is noisy and hard to integrate accurately without accumulating drift. In contrast, it is easy to detect movement with an accelerometer: within a stable (spike-free) period, the accelerometer shows a significantly higher variance while moving than when stationary.

Accordingly, we compute a boolean (true/false) movement hint for each time slot. We divide the data into one-second slots and compute the standard deviation of the 3-axis magnitude of the acceleration in each slot. Directly thresholding standard deviation sometimes results in spurious detections when the vehicle is static and the signal exhibits a short-lived outlier. To fix this, we apply an EWMA filter to the standard deviation stream to remove short-lived outliers. We then apply a threshold $\sigma_{movement}$, on the standard deviation to label each time slot as "static" or "moving". We used a subset of our driving data across multiple phones as training (where we do know ground truth from GPS), to learn the optimal value of $\sigma_{movement}$, which turned out to be approximately 0.15 ms^{-2} for one-second windows. Figure 6 illustrates our movement hint extraction algorithm on example data.

Turn Hints. The orientation sensor of a smartphone (compass/gyroscope) provides orientation about three axes. We are interested in the axis that provides the relative rotation of the phone about an axis parallel to gravity (called "yaw" on the iPhone 4).

Because the phone can be in any orientation to start with in a handbag or pocket, we do not use the absolute orientation in any of our algorithms. We have observed that irrespective of how the phone is situated, a true change in orientation manifests as a *persistent*, *significant*, and *steep* change in the value of the orientation sensor.

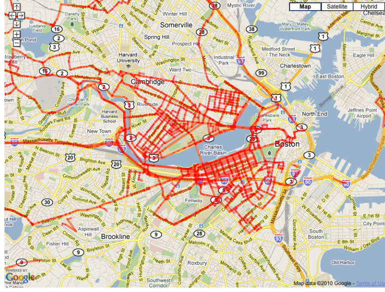


Figure 8: Coverage map of our driving data set.

With compass data, the main challenge is that the orientation reported is noisy because metallic objects nearby, or because the compass becomes uncalibrated. We solved this problem by applying a *median filter* with a three-second window on the raw orientation values, which filtered out non-persistent noise with considerable success (a mean filter would also remove noise, but would blur sharp transitions that we do want to observe). We then find transitions with a magnitude exceeding at least 20 degrees and slope exceeding a threshold, which we fixed at 1.5 by experimentation.

Figure 7 illustrates a plot of the compass data with the turn marked, and the processing steps required to generate a turn hint. We note that even after filtering, a true change in orientation can sometimes be produced by the phone sliding around within a pocket or a bag, or turning for reasons other than the car actually turning.

6 Evaluation

In this section, we show that the trajectory matches produced by *CTrack* are: (1) accurate enough to be useful for various tracking and positioning applications, (2) superior to sub-sampled GPS in terms of the accuracy-energy tradeoff, and (3) significantly better than strategies that reduce cellular fingerprints to point locations before matching. We investigate how much each of the four techniques used in CTrack —sequencing, windowing, smoothing, and sensor hints—contribute to the gains in accuracy.

6.1 Method and Metrics

We evaluate CTrack on 126 hours of real driving data in the Cambridge-Boston area, collected from 15 Android G1 phones and one Nexus One phone over a period of 4 months. We configured our phone library for the Android OS to continuously log the ground truth GPS location and the cell tower fingerprint every second, and the accelerometer and compass at 20 Hz. Our data set covers 3,747 road segments, amounts to 1,718 km of driving, and 560 km of distinct road segments driven. The data set includes sightings of 857 distinct cell towers. Figure 8 shows a coverage map of the distinct road segments driven in our data set.

From 312 drives in all, we selected a subset of 53 drives verified manually to have high GPS accuracy as *test drives*, amounting to 109 distinct km. We picked a limited subset as test drives to ensure each test drive was contained entirely within a small bounding box with dense training coverage. This is because evaluating the algorithm in areas of sparse coverage (which many of the other 259 drives venture into) could bias results in our favor by reducing the number of candidate paths to map-match to. For each test drive, we perform *leave-one-out* evaluation of the map-matching algorithm: we train our algorithm on all 311 drives excluding the test drive, and then map-match the test drive using CTrack. We do this to ensure enough training data for each drive, and at the same time to keep the evaluation fair.

We compare CTrack to two other strategies in terms of energy and accuracy:

1. *GPS* k gets one GPS sample every k minutes ($k = 2,4$), interpolates, and map-matches it using VTrack [32].

2. *Placelab-VTrack* computes the best static localization estimate for each time instant using Placelab's technique [8], and matches the static estimates using VTrack [32]. The VTrack paper shows that its HMM does much better than just matching each point to the nearest segment.

We use three metrics in our evaluation of accuracy: *precision*, *recall*, and *geographic error*. Our precision and recall are similar to conventional precision and recall, but take the order of matched segments in the trajectory into account. We say that a subset of segments in a trajectory T_1 that also appears in trajectory T_2 are *aligned* if those segments appear in T_1 in the same order in which they appear in T_2. Given a ground truth sequence of segments G and an output sequence X to evaluate (produced by one of the algorithms), we run a dynamic program to find the maximum length of aligned segments between G and X. We define:

$$Precision = \frac{Total\ length\ of\ aligned\ segments}{Total\ length\ of\ X} \tag{2}$$

$$Recall = \frac{Total\ length\ of\ aligned\ segments}{Total\ length\ of\ G} \tag{3}$$

We estimated the ground truth sequencing of segments by map-matching GPS data sampled every second with VTrack [32], and manually fixing a few minor flaws in the results.

Geographic Error. Precision and recall are relevant to applications that care about obtaining information at a segment-level, such as traffic monitoring. However, applications such as visualization do not need to know the exact road segments traversed, but may want to identify the broad contours of the route followed (e.g., mistaking a road for a nearby parallel road may be acceptable).

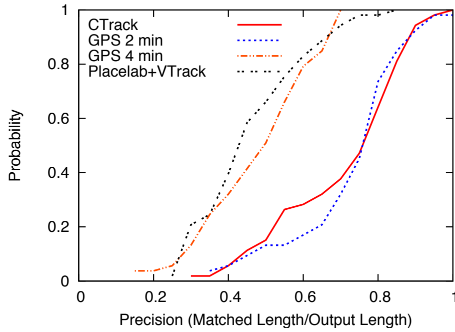


Figure 9: CDF of Precision: Comparison.

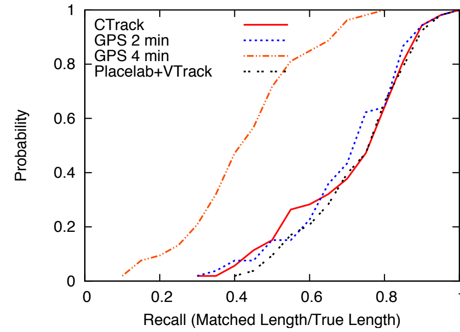


Figure 10: CDF of Recall: Comparison.

To quantify this notion, we compute a third metric, *geographic error*, which captures the spatial distance between the ground truth and the matched output. We compute the maximum alignment between the ground truth trajectory G and output trajectory X using dynamic programming. This alignment matches each segment S of X to either the same segment S on G (if CTrack matched that segment correctly) or to a segment $S_{wrong} \in G$ (if matched incorrectly). Define the *segment geographic error* to be the distance between S and S_{wrong} for incorrect segments, and 0 for correctly matched segments. The mean segment geographic error over all segments in X is the *overall geographic error*.

6.2 Key Findings

The key findings of our evaluation are:

1. CTrack has 75% precision and 80% recall in both the mean and median, and a median geographic error of 44.7 meters. We discuss what these numbers mean in the context of real applications below.

2. CTrack has 2.5× better precision and 3.5× smaller geographic error than *Placelab+VTrack*.

3. CTrack is equivalent in precision to map-matching GPS sub-sampled every 2 minutes while consuming over 2.5× less energy. It also reduces error ($1 - precision$) by a factor of over 2× compared to sub-sampling GPS every 4 minutes, consuming a similar amount of energy. CTrack is 6× better than continuous WiFi sampling in terms of battery lifetime on the Android platform.

4. The first step of CTrack, grid sequencing, is critical. Without sequencing, CTrack effectively reduces to computing a (lat, lon) estimate from the best fingerprint match, ignoring all other data. The median precision without sequencing is only 50%. See Section 6.4 for more detail.

5. We can extract movement and turn hints from raw sensor data with approximately 75% precision and recall. These hints improve accuracy by removing spurious loops and turns in the output. Using hints improves precision by 6% and recall by 3%. See Section 6.5 for more detail.

6.3 Accuracy Results

Figure 9 shows a CDF of the map-matching precision for CTrack, *GPS k* (for $k = 2, 4$ minutes) and *Placelab+VTrack*. CTrack has a median precision of 75%, much higher than the both the energy-equivalent strategy of sub-sampling GPS every 4 minutes (48%), and *Placelab+VTrack* (42%). In effect, CTrack has over 2× lower error ($1 - precision$) than sub-sampling GPS every 4 minutes, and over 2.5× lower error than map-matching cellular localization estimates output by the Placelab method. Also, CTrack has equivalent precision to map-matching GPS sub-sampled every two minutes, while reducing energy consumption by approximately 2.5× compared to this approach (Figure 2).

Figure 10 shows a CDF of the recall. All the strategies except *GPS 4* min are equivalent in terms of recall. Sub-sampling GPS every four minutes has poor recall (median only 41%) because a four-minute sampling interval misses significant turns in our input drives and finds the wrong path. The fact that *Placelab+VTrack* has identical recall shows that simple static cellular localization does manage to recover a significant part of the input drive. However, converting cellular fingerprints directly to points results in significant noise and long-lived outliers, and hence produces a large number of incorrect segments when map-matched directly (i.e., has low precision).

To understand what 75% precision might mean in terms of a an actual application, we refer readers to our work on VTrack [32], which studies the relationship between map-matching accuracy and the accuracy of two end-to-end applications: traffic delay monitoring and traffic hot-spot detection. We found that a median precision of 85% was still useful for accurate traffic delay estimation. Our results for cellular (75%) are only somewhat worse, and while not directly comparable, they suggest a significant portion of delay data from CTrack would be useful.

For applications such as route visualization, or those that aggregate statistics over paths (e.g., to compute histograms over which of n possible routes is taken), or

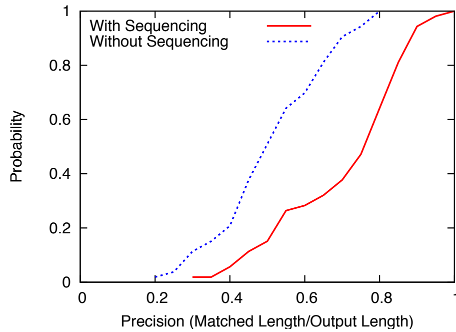


Figure 11: Precision with and without grid sequencing.

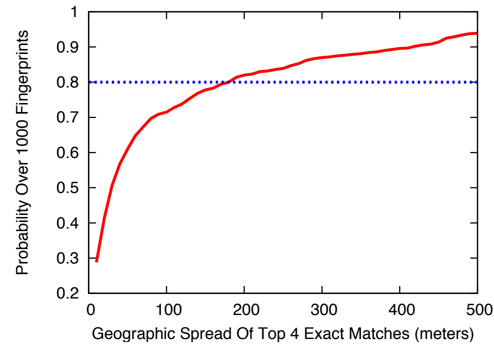


Figure 12: Geographic spread of exact matches. The dashed line shows the 80th percentile.

those that simply show a user's location on a map, getting most segments right with a low overall error is likely sufficient. Our median geographic error is quite low—just 45 meters—suggesting CTrack would have sufficient accuracy for such applications. In contrast, the median geographic error of the Placelab+VTrack approach is 156 meters, over 3.5× worse than CTrack.

Filtering using a confidence predictor. We investigated whether a confidence metric could be used to filter out drives on which CTrack does poorly, thereby trading-off some recall for substantially better precision, which would be useful for accuracy-sensitive applications. We found two predictors, both weakly correlated with map-matching accuracy: (a) the 90th percentile distance of smoothed grids from the segments they are matched to, and (b) the mean difference (over all points P) in emission score between the segment that P is matched to in the output, and the segment closest to P. The intuition is that a point far away from the road segment it is matched to, or closer to a different road segment, implies lower confidence in the match. When applying these confidence filters to our output drives, we currently improve the median precision from 75% to 86%, but lose substantially in terms of recall, whose median reduces from 80% to 35%). In future work, we plan to explore whether *boosting* [12] can combine these weak confidence predictors into a stronger one.

6.4 Benefit of Sequencing

We elaborate on one of our key technical contributions: the idea that the first pass of grid sequencing *before* converting fingerprints to geographic locations is crucial to achieving good matching accuracy. We provide experimental evidence supporting this idea. We also show that windowing and smoothing help improve matching accuracy, though to a lower degree.

Impact of Sequencing. Figure 11 is a CDF that compares the precision of CTrack with and without the first pass of grid sequencing. This figure shows that sequencing is critical to achieving reasonable accuracy: without sequencing, the median precision drops from 75% to 50%. The reason is that running CTrack without sequencing amounts to reducing each fingerprint to its best match in the training database, ignoring the sequence of points.

As mentioned earlier, reducing a fingerprint to a single geographic location loses information because a given cellular fingerprint is seen from multiple locations quite far apart. Figure 12 illustrates the CDF of this geographic spread. We selected 1000 fingerprints at random from our training data. For each fingerprint F, we found all the *exact matches* for F, i.e. fingerprints F' with the exact same set of towers in the training data as F. We ordered the matches by similarity in signal strength, most similar first, and computed the geographic diameter of the top k matches for each fingerprint (using $k = 4$).

The figure shows that over 20% of matching sets have a diameter exceeding 150 meters, and at least 10% have a diameter exceeding 400 meters. Recall that the methods in Placelab (and RADAR, if applied to cellular data) would simply compute the centroid of the top k matches. This approach does not work well for sets with a large geographic spread, and motivates the need for the fundamentally different approach used in CTrack in which we keep track of *all* possible likely locations, and then use a continuity constraint to *sequence* these locations in two steps.

Windowing and Smoothing. Table 1 shows the precision and recall of CTrack with and without windowing and smoothing, two other heuristics used in CTrack. We see that each of these features improves the precision by approximately 10%, which is a noticeable quantity. The recall does not improve because the algorithm without windowing/smoothing is good enough to identify most of the segments driven: the heuristics mainly help eliminate loops in the output.

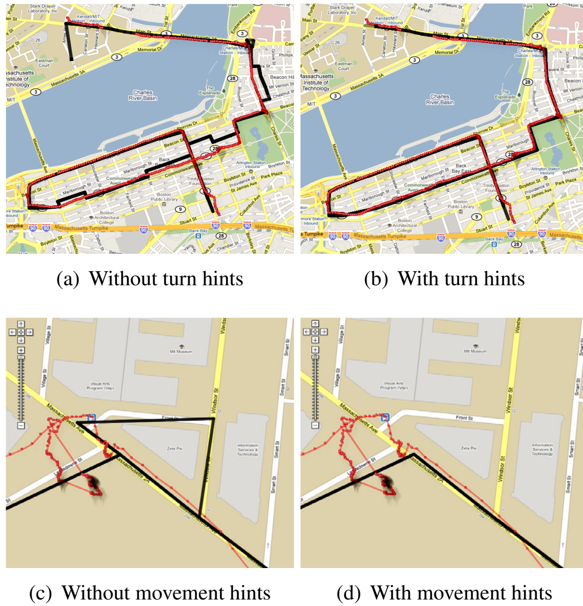
6.5 Do Sensor Hints Help?

Figure 13 illustrates by example how turn hints extracted from the phone compass help in trajectory matching. Without using turn hints (Figure 13(a)), our algorithm

	With		Without	
	Prec.	Recall	Prec.	Recall
Windowing	75.4%	80.3%	65.6%	82.3%
Smoothing	75.4%	80.3%	66.5%	82.5%

Table 1: Windowing and smoothing improve median trajectory matching precision.

finds the overall path quite accurately but includes several spurious turns and kinks, owing to errors in cellular localization. After including turn hints in the HMM, the false turns and kinks disappear (Figure 13(b)).



(a) Without turn hints (b) With turn hints

(c) Without movement hints (d) With movement hints

Figure 13: Sensor hints from the compass and accelerometer aid map-matching. Red points show ground truth and the black line is the matched trajectory.

In Figure 13(c), the driver stopped at a gas station to refuel, which can be seen from the cluster of ground-truth GPS points. Before using movement hints, errors from cellular localization were spread out, causing the map-matching to introduce a loop not present in the ground truth (Figure 13(c)). After incorporating movement hints, the speed constraint in our HMM eliminates this loop because it detects that the car would not have had sufficient time to complete the loop (Figure 13(d)). We note a limitation of the movement hint: this kind of stop detection works because the phone was placed on the dashboard: if it had been in the driver's pocket during refueling, the movement hints would not have helped had the driver gotten out of the car and been moving about.

Figure 14 is a CDF that compares the precision of CTrack with and without sensor hints (both movement and turn). This figure shows that sensor hints improve the median precision of matching by approximately 6%. While this may not seem huge, there exist several trajec-

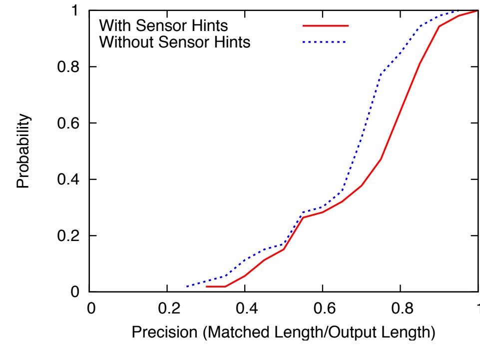


Figure 14: Precision with and without sensor hints.

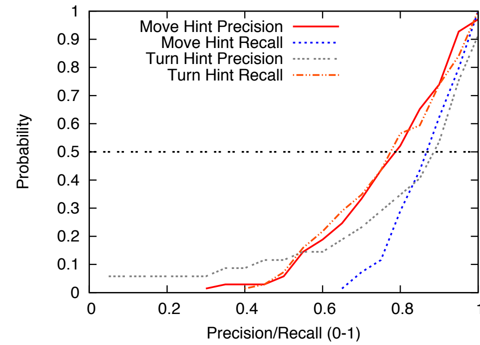


Figure 15: Precision/Recall CDF For Hint Extraction.

tories for which the hints do help significantly, suggesting that using them is a good idea when available. In our experience, the main benefit of the hints is in eliminating the several "kinks" and spurious turns in the matched trajectory, which our metrics don't adequately capture.

We used the ground truth GPS to measure how accurately our CTrack is able to extract individual movement and turn hints. We found that the median precision and recall of both motion and turn hint extraction exceeds 75%.

6.6 How Much Training?

To quantify the amount of training data essential to achieving good trajectory mapping accuracy with CTrack, we picked a pool of test drives at random, amounting to 5% of our data set (8 hours of data), and designated the remaining 95% as the training pool. We picked subsets of the training pool of increasing size, i.e. first using fewer drives for training, then using more. In each run, the training subset was used to train CTrack and then evaluated on the test pool. Figure 16 shows the mean precision and recall of CTrack on the test pool as a function of the number of drive hours of training data used to train the system. The accuracy is poor for very small training pools, as expected, but encouragingly, it quickly increases as more training data is available. The algorithm performs almost as accurately with 40 hours of training data as with 120, suggesting that 40 hours of training is sufficient for our data set.

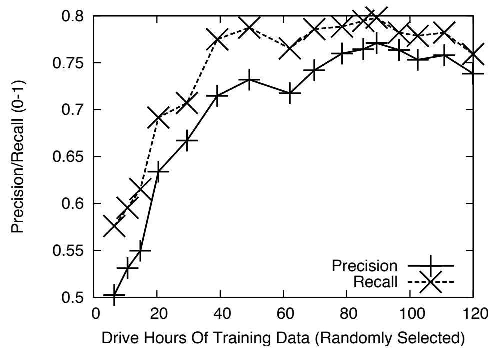


Figure 16: Prec./Recall vs Training Data Size.

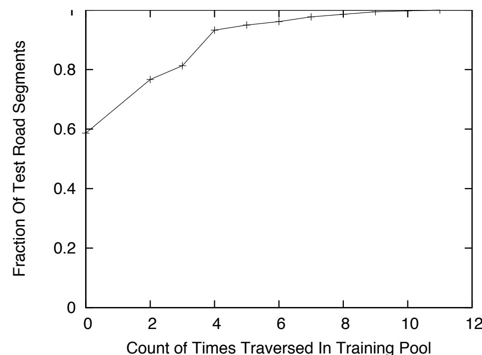


Figure 17: CDF of Drive Counts, 40 Hrs Training Data.

The 40-hour number, of course, is specific to the geographic area we covered in and around Boston, and to the test pool. To gain more general insight, we measure the *drive count* for each road segment in the test pool, defined as the number of times the segment is traversed by any drive in the training pool. Figure 17 shows the distribution of *test segment drive counts* corresponding to 40 hours of training data. While the mean drive count is approximately 3, this does not mean each road segment on the map needs to be driven thrice to achieve good accuracy. As the graph shows, about 60% of the test segments were not traversed even once in the training pool, but we can still map-match many of these segments correctly. The reason is that they lie in the same grid cell as some nearby segment that was driven in the training pool. This result promising because it suggests that training does not have to cover every road segment on the map to achieve acceptable accuracy.

7 RELATED WORK

Placelab performed a comprehensive study of GSM localization and used a fingerprinting scheme for cellular localization [8]. RADAR used a similar fingerprinting heuristic for indoor WiFi localizations [3], and our map-matching emission score is inspired by these methods. However, neither Placelab nor RADAR address the problem of trajectory matching, and are concerned with the accuracy of individual localization estimates, rather than finding the optimal sequencing of estimates. As shown by our results, this sequencing step is critical: applying a map-matching algorithm directly to Placelab-style location estimates results in significantly worse accuracy (by a factor of over $2\times$) compared to CTrack.

Letchner et al. [17] and our previous work on VTrack [32] use HMMs for map-matching. However, these previous algorithms use and process (lat, lon) coordinates as input and use a Gaussian noise model for emissions, and are hence unsuitable and inaccurate for map-matching cellular fingerprints, as shown by our results. Nor do they use sensor hints.

CompAcc [10] proposes to use smartphone compasses and accelerometers to find the best match for a walking trail by computing directional "path signatures" for these trails. They do not use cell towers. However, from our understanding, the paper uses absolute values of compass readings. This approach did not work in our experiments, because the absolute orientation of a phone can be quite different depending on whether it is in a driver's pocket, on a flat surface, or held in a person's hand. For this reason, we chose to use boolean turn hints instead, which are more robust and can be accurately computed regardless of changes in the phone's initial orientation or position. For extracting motion hints and detecting walking and driving using the accelerometer, we use algorithms similar to those in [27, 31, 26].

Some previous papers [9, 23, 16] have proposed energy-efficient localization schemes that reduce reliance on continuously sampling GPS by using a more energy-efficient sensor, such as the accelerometer, to trigger sampling GPS. RAPS [23] also uses cell towers to "blacklist" areas where GPS accuracy is low and hence GPS should be switched off, to save energy. However, none of these papers address trajectory matching or propose a GPS-free, accurate solution for map-matching.

Skyhook [29] and Navizon [20] are two commercial providers for WiFi and Cellular localization, providing databases and APIs that allow programmers to submit WiFi access point(s) or cell tower(s) and look up the nearest location. However, to the best of our knowledge, they do not use any form of sequencing or map-matching, and focus on providing the best static localization estimate.

8 CONCLUSION

We described CTrack, an energy-efficient, GPS-free system for trajectory mapping using cellular tower fingerprints. The key lesson we learned was that sequencing cellular fingerprints before matching them is critical to achieving good accuracy. On smartphones, our CTrack implementation uses close to zero extra energy while achieving good mapping accuracy, making it a good way to distribute collaborative trajectory-based applications

like traffic monitoring to a huge number of users without any associated energy consumption or battery drain concerns. A GPS-free approach to trajectory matching also opens up the possibility of providing more fine-grained location services on the world's most popular, cheapest phones that do not have GPS, but that do have GSM connectivity.

Acknowledgments

This work was supported by the National Science Foundation under grant CNS-0931550.

References

[1] Analog Devices AD9864 Datasheet: GSM RF Front End and Digitizing Subsystem. http://www.analog.com/static/imported-files/data_sheets/AD9864.pdf.

[2] Analog Devices, Inc. *ADXL330: Small, Low Power, 3-Axis +/-3 g iMEMS Accelerometer (Data Sheet)*, 2007. http://www.analog.com/static/imported-files/data_sheets/ADXL330.pdf.

[3] P. Bahl and V. Padmanabhan. RADAR: An In-building RF-based User Location and Tracking System. In *INFOCOM*, 2000.

[4] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications. In *IMC*, 2009.

[5] F. Ben Abdesslem, A. Phillips, and T. Henderson. Less is more: Energy-efficient Mobile Sensing with SenseLess. In *MobiHeld*, 2009.

[6] GPS and Mobile Handsets. http://www.berginsight.com/ReportPDF/ProductSheet/bi-gps4-ps.pdf.

[7] A. Boustani, L. Girod, D. Offenhuber, R. Britter, M. I. Wolf, D. Lee, S. Miles, A. Biderman, and C. Ratti. Investigation of the Waste Removal Chain Through Pervasive Computing. *IBM Journal of Research and Development*, 2010.

[8] M. Y. Chen, T. Sohn, D. Chmelev, D. Haehnel, J. Hightower, J. Hughes, A. Lamarca, F. Potter, I. Smith, and A. Varshavsky. Practical Metropolitan-scale Positioning for GSM Phones. In *UbiComp*, 2006.

[9] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox. EnLoc: Energy-Efficient Localization for Mobile Phones. In *INFOCOM*, 2009.

[10] I. Constandache, R. Roy Choudhury, and I. Rhee. CompAcc: Using Mobile Phone Compasses and Accelerometers for Localization. In *INFOCOM*, 2010.

[11] Fedex intros Senseaware Sensor for Tracking Packages. http://www.electronista.com/articles/09/11/27/senseaware.sensor.sends.temps.drops.more.

[12] Y. Freund and R. E. Schapire. A Decision Theoretic Generalization of Online Learning and an Application to Boosting. In *EuroCOLT*, 1995.

[13] S. Gaonkar, J. Li, R. R. Choudhury, L. Cox, and A. Schmidt. Micro-Blog: Sharing and Querying Content through Mobile Phones and Social Participation. In *MobiSys*, 2008.

[14] Information On Human Exposure To Radiofrequency Fields From Cellular and PCS Radio Transmitters. http://www.fcc.gov/oet/rfsafety/cellpcs.html.

[15] iCartel. http://icartel.net/icartel-docs/.

[16] M. B. Kjærgaard, J. Langdal, T. Godsk, and T. Toftkjær. EnTracked: Energy-efficient Robust Position Tracking for Mobile Devices. In *MobiSys*, 2009.

[17] J. Krumm, J. Letchner, and E. Horvitz. Map Matching with Travel Time Constraints. In *SAE World Congress*, 2007.

[18] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao. Energy-accuracy Trade-off for Continuous Mobile Device Location. In *MobiSys*, 2010.

[19] LoJack Car Security System For Stolen Vehicle Recovery. http://www.lojack.com.

[20] Navizon. http://www.navizon.com.

[21] Qualcomm Transportation: OmniTRACKS Mobile Communications System. http://www.qualcomm.com/products_services/mobile_content_services/enterprise/omnitracs.html.

[22] OpenStreetMap. http://www.openstreetmap.org.

[23] J. Paek, J. Kim, and R. Govindan. Energy-efficient Rate-adaptive GPS-based Positioning for Smartphones. In *MobiSys*, 2010.

[24] PNI Corporation. *MicroMag3 3-Axis Magnetic Sensor Module*. http://www.sparkfun.com/datasheets/Sensors/MicroMag3%20Data%20Sheet.pdf.

[25] Qualcomm inGeo Service. http://www.qualcomm.com/innovation/stories/ingeo.html.

[26] L. Ravindranath, C. Newport, H. Balakrishnan, and S. Madden. Improving Wireless Network Performance Using Sensor Hints. In *NSDI*, 2011.

[27] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using Mobile Phones to Determine Transportation Modes. *Transactions on Sensor Networks*, 6(2), 2010.

[28] RunKeeper. http://runkeeper.com.

[29] Skyhook. http://www.skyhookwireless.com.

[30] Telit GE865 Datasheet. http://www.telit.com/module/infopool/download.php?id=1666.

[31] A. Thiagarajan, J. Biagioni, T. Gerlich, and J. Eriksson. Cooperative Transit Tracking Using GPS-enabled Smart-phones. In *SenSys*, 2010.

[32] A. Thiagarajan, L. Sivalingam, K. LaCurts, S. Toledo, J. Eriksson, S. Madden, and H. Balakrishnan. VTrack: Accurate, Energy-Aware Road Traffic Delay Estimation Using Mobile Phones. In *SenSys*, 2009.

[33] TomTom. http://www.tomtom.com.

[34] Trash Track. http://senseable.mit.edu/trashtrack.

[35] A. J. Viterbi. Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm. In *IEEE Transactions on Information Theory*, 1967.

[36] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A Framework of Energy Efficient Mobile Sensing for Automatic User State Recognition. In *MobiSys*, 2009.

Improving Wireless Network Performance Using Sensor Hints

Lenin Ravindranath, Calvin Newport, Hari Balakrishnan and Samuel Madden
MIT Computer Science and Artificial Intelligence Laboratory
{lenin, cnewport, hari, madden}@csail.mit.edu

Abstract

With the proliferation of mobile wireless devices such as smartphones and tablets that are used in a wide range of locations and movement conditions, it has become important for wireless protocols to adapt to different settings over short periods of time. Network protocols that perform well in static settings where channel conditions are relatively stable tend to perform poorly in mobile settings where channel conditions change rapidly, and vice versa. To adapt to the conditions under which communication is occurring, we propose the use of *external sensor hints* to augment network protocols. Commodity smartphones and tablet devices come equipped with a variety of sensors, including GPS, accelerometers, magnetic compasses, and gyroscopes, which can provide hints about the device's mobility state and its operating environment. We present a wireless protocol architecture that integrates sensor hints in adaptation algorithms. We validate the idea and architecture by implementing and evaluating sensor-augmented wireless protocols for bit rate adaptation, access point association, neighbor maintenance in mobile mesh networks, and path selection in vehicular networks.

1 Introduction

With over 172 million devices sold in 2009, smartphones are a rapidly growing market [27]. Some analysts predict that smartphones and pads/tablets will surpass worldwide PC sales by the end of 2011 [20]. These devices may well become the dominant mode of Internet access in the near future [19].

With the proliferation of these "truly mobile" devices, it is increasingly common for wireless network protocols to have to deal with *both* static and mobile usage within a short time period. Consider, for example, a smartphone user at the supermarket who alternates between standing still in front of product displays and moving between aisles, all the while streaming audio through the in-store wireless network. Mobility introduces difficult problems that wireless network protocols must overcome to achieve good performance. During motion, the vagaries of wireless communication become more pronounced: channel quality varies rapidly, losses become more bursty, and assessments of channel behavior are quickly outdated. Because of this, nodes should not maintain long histories, as the rapidly changing channel conditions and network topology would quickly render them invalid. Routing tables may also need to adapt quickly to neighbor changes, and the optimal next-hop may depend on the direction and speed of movement.

However, strategies that compensate for these mobility-related difficulties are unlikely to be optimal in stationary scenarios [4, 25]. When nodes are static, they can average estimates of channel quality, observe their neighbors, and compute routes over long time scales (many seconds), carefully obtaining and updating observations from many packets. In so doing, they can correctly avoid reacting to the inevitable short-term variations that even static wireless networks encounter (e.g., due to short-term fading). Previous work has generally not distinguished between these modes, attempting instead to adapt seamlessly across extremely different network conditions.

The key insight in our work is that nodes can use external (to the network stack) *sensor hints* to improve the performance of wireless network protocols. Our approach is practical and readily implementable because almost every smartphone and tablet today comes equipped with a wide array of sensors like GPS, accelerometers, compasses, and so on. These sensors are used by applications, but are largely ignored by the network stack and protocols. We show how data from these sensors can provide hints to protocols about the *mobility mode* of the device. By "mobility mode," we mean attributes such as whether the device has started moving or is static, its speed of motion, its position, and the heading (direction) of motion—all factors that affect wireless network protocol performance. Protocols can explicitly adapt their behavior and parameters to the current mobility mode.

Sensor hints may be used in different ways in different protocols. When a node generates a hint locally or receives a hint from a neighbor, it may adapt in response to it. The adaptation might be continuous in nature (e.g., updating protocol parameters) or discrete (e.g., switching from a static-optimized to a mobility-optimized protocol). In Section 2, we introduce a novel sensor-augmented wireless architecture that allows de-

vices to extract hints and provide them to protocols. To the best of our knowledge, ours is the first general approach to using sensor hints to augment a variety of network protocols.

In addition to the sensor-augmented network architecture, we make four contributions:

1. Hint-aware bit rate adaptation: In Section 3, we describe and evaluate our implementation of a novel frame-based bit rate adaptation protocol, *RapidSample*, and show through trace-based simulation and testbed experiments that it obtains up to 70% better throughput than existing frame-based and SNR-based rate adaptation protocols, and comparable throughput to SoftRate [25], *when a node is in motion*. We use RapidSample to develop a hint-aware bit rate adaptation protocol that switches strategies based on mobility hints and show through exhaustive trace-based evaluation and testbed experiments that it obtains between 17% and 52% better throughput than SampleRate, 17% and 39% better throughput than RRAA, and 11% and 47% better throughput than SNR-based schemes, in mixed mobility scenarios in various environments.

2. WiFi access point (AP) association: In Section 4, we describe a hint-aware AP association protocol with two modes: maximizing bulk transfer throughput and minimizing handoffs. We show through trace-based evaluation that the hint-aware protocol improves throughput by 30% and reduces the number of handoffs by 40% compared to today's standard scheme.

3. Mobile topology maintenance: In Section 5, we show experimentally that maintaining acceptable error rates for topology maintenance while mobile requires over 20 times more traffic than in the stationary case. We implement a hint-aware protocol that switches to this expensive probing only when in motion.

4. Path selection in vehicular mesh networks: In Section 6, we present a collection of hint-aware path selection metrics for vehicular networks and show, using trace-based simulation, that they increase the stability of short routes by nearly a factor of 5 compared to the hint-free approach.

2 Design

Current wireless protocols adapt their behavior based on *in-network* information such as loss rate, bit errors, or SNR. In contrast, we present a hint-aware protocol architecture that augments this in-network information with hints from external sensors, which can be used at all layers of the network stack to improve performance. In addition to using local sensor hints, a protocol can also adapt based on sensor hints communicated from other nodes.

In this section, we first present a general-purpose hint-aware protocol architecture. We then describe simple and

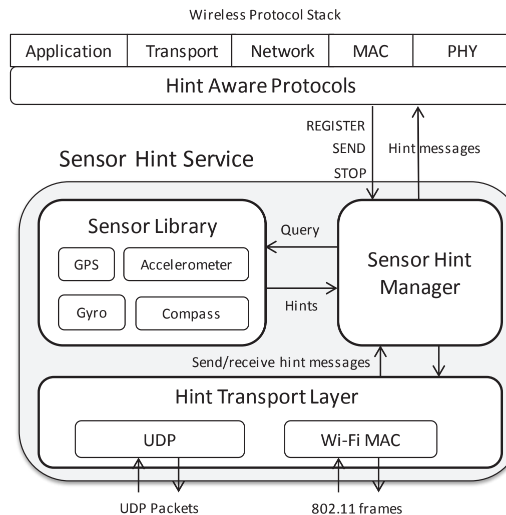


Figure 1: Hint-aware protocol architecture.

Hint Type	Hint Value
Movement	True/False
Walking	True/False
Heading	Degrees Relative to True North
Speed	Miles per Hour
Environment	Indoor/Outdoor

Figure 2: Hint types exposed by the *Sensor Library*.

accurate techniques for extracting mobility hints from sensors such as GPS, accelerometers and compasses.

2.1 Hint-Aware Protocol Architecture

Figure 1 depicts the architecture; the goal is to make it easy to augment wireless network protocols with sensor hints. The architecture provides a *Sensor Hint Service* that abstracts and hides the details of (1) querying various sensors, (2) extracting hints from raw sensor data, and (3) communicating relevant hints over the network. The service exposes well-defined interfaces to achieve these goals. Our current implementation of the Sensor Hint Service runs as a background service on the Android platform and as a Click module for Linux mobile devices. It should be straightforward to incorporate this service into other mobile platforms.

The Sensor Hint Service has three components:

1. Sensor Library. The Sensor Library processes raw sensor data to extract useful hints. We focus on mobility hints and our implementation currently supports the hint types shown in Figure 2. Section 2.2 discusses how these hints are extracted.

2. Hint Transport Layer. Some protocols can benefit from hints from other nodes. For instance, a bit rate adaptation protocol can adapt its bit rate using not only its own movement hints, but also movement hints from nodes the protocol is communicating with. The Hint

Transport Layer provides a protocol-independent way to communicate hints.

When sending a hint to another node, the *Sensor Hint Manager* (described below) constructs a hint message (shown in Figure 3) and delivers it to the Hint Transport Layer, which in turn sends the hint. The hint message consists of the source MAC address and ⟨hint type, hint value⟩ pairs. When receiving a hint from another node, the Hint Transport Layer delivers the received hint message to the Sensor Hint Manager, which in turn delivers it to the appropriate protocol.

The Hint Transport Layer provides two communication mechanisms to send and receive hints. The first uses UDP. Each node opens a pre-defined UDP port, the HINTS port, to receive hint messages. Hint messages may either be unicast or broadcast to this UDP port.

The UDP scheme works only as long as the nodes are connected through IP. In certain hint-aware wireless protocols (Section 5 and Section 6) nodes do not have IP connectivity, instead communicating via a link-layer protocol such as 802.11's link layer. Thus, for our second scheme, we use a reserved protocol type in the link-layer MAC header to denote a hint message frame (Figure 3). The Hint Transport Layer then listens for unicast or broadcast hints sent in link-layer frames. An alternative scheme might be to overload or piggy-back hints on existing 802.11 frames; we leave the exploration of this possibility to future work.

Because Android phones do not (yet) support sending raw 802.11 frames from user-level, we implemented only the UDP mechanism for phones. For Linux devices, we implemented both schemes. Legacy nodes not running the Sensor Hint Service will simply ignore the hint messages, as long as the HINTS port is not in use by some other application.

3. Sensor Hint Manager. The Sensor Hint Manager arbitrates communication between the protocol, the Sensor Library and the Hint Transport Layer. It exposes a local socket interface (different from the HINTS port) for protocols to interact with the Sensor Hint Service. Protocols register for one or more hints using REGISTER (HintTypes[], ReportRate, CallbackPort, Source). Once registered, the Sensor Hint Manager uses the CallbackPort to stream hints to the protocol. The Source field can be LOCAL, REMOTE, or ALL, corresponding to local hints, remote hints, or both. The protocol can specify a ReportRate, in milliseconds, which indicates how often to report the hint. ReportRate also takes two special values: "0" means "as fast as possible" and -1 means "only when there is a change in the hint state".

Protocols use SEND(HintTypes[], SendRate, ComType, Address) to instruct the service to send hints to other nodes. SendRate takes values similar to ReportRate in the REGISTER command, with the same conventions. ComType specifies the communication types (currently either UDP or MAC frames). Hints may be unicast to a specific node or broadcast in either ComType setting.

REGISTER and SEND both return a unique ID to the protocol. The protocol can use the returned ID to stop sending hints using the STOP (ID) command.

2.2 Extracting Hints

In this section, we describe how to extract the hints shown in Figure 2—*movement*, *walking*, *heading*, *speed*, and *environment*—using standard sensors found on most smartphones and tablets.

Movement hint. Movement is a boolean hint that is true if, and only if, a device is moving, i.e., if either the device's acceleration or its speed is non-zero. We obtain this information from the acceleration sensor indoors, and from the combination of GPS and the acceleration sensors outdoors. Note that it is important to quickly capture the situation when a device has started moving after being at rest, and vice versa, so measuring the acceleration is important.

The accelerometer on most smartphones reports force values for its x, y, and z axes, at a certain sample rate (usually 20–500 Hz). The values are reported either in $\mathrm{m/s^2}$ or in terms of g ($= 9.8\ \mathrm{m/s^2}$). Figure 4 plots a raw accelerometer trace of a smartphone user who walks in the 6–14 second and 22–32 second periods, and is static the rest of the time. The accelerometer shows a significantly higher variance while moving than when stationary. We use this variance to extract a movement hint.

For every new accelerometer sample, we compute the standard deviation of the magnitude of the acceleration over a sliding window (w) of samples. The window slides by one sample for each computation. If the standard deviation in a window exceeds a threshold (a), we detect movement. When the standard deviation is within the threshold for n successive sliding windows, we report that the node is stationary.

We experimented with many values for w, a, and n and determined that $w = 5, a = 0.15\ \mathrm{m/s^2}$, and $n = 10$ gave us few false hints. Figure 5 illustrates our movement hint extraction for the trace in Figure 4. We have implemented the above technique on four different platforms (Android Nexus one, Android Google G1, iPhone 4 and SparkFun accelerometer that connects to a Linux laptop) and found that the parameters offer good performance in all cases.

On the Android platform with a maximum accelerometer sample rate of 50 Hz, we were able to detect movement within 100 ms and detect that the node became stationary within 200 ms. On the Sparkfun platform, with a sample rate of 500 Hz, we were able to detect movement within 10 ms and stationarity within 20 ms.

The movement hint is used by the protocols described

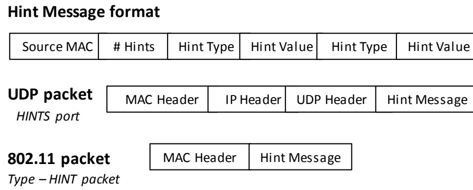


Figure 3: Hint message and packet formats.

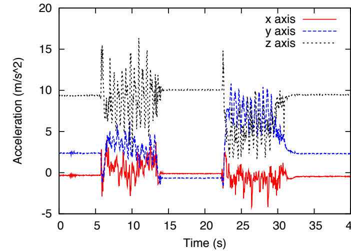


Figure 4: Raw accelerometer trace.

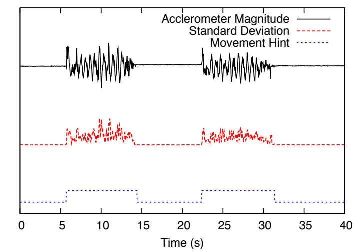


Figure 5: Movement hint extraction from accelerometer data.

in Section 3 and Section 5 to improve bit rate adaptation and topology maintenance, respectively.

Walking hint. Whereas a simple movement hint is useful in some cases, in other situations it is valuable to detect whether a user is walking versus other types of movement, such as when the user is stationary but moving the device. We accomplish this using the walking detector developed in TransitGenie [22] and apply it to AP selection (Section 4).

Heading hint. Heading can be determined from digital compasses (magnetometers) that are available on many devices. GPS also allows us to infer a heading when a device is moving outdoors. These sensors produce a heading in degrees relative to the earth's magnetic north pole. To use a compass to determine the heading of the user holding a device, and not the heading of the device itself, it is necessary to first determine the device's orientation. The standard technique used by inertial navigation systems is to use gyroscope sensors in conjunction with the accelerometer to infer this orientation [21]. In our indoor experiments, we assume we know the orientation of the device, and use only the compass readings. These heading hints are used by the protocols described in Section 4 and Section 6 to improve access point selection and vehicular path selection, respectively.

Speed hint. To determine a speed hint outdoors we can use the speed values reported by GPS. We use this hint in Section 6 for path selection.

Environment (indoor/outdoor) hint. To determine whether a user is indoors or outdoors we use the fact that it is typically impossible to get a GPS fix indoors. In Section 4 we use this hint to improve AP association.

3 Hint-Aware Bit Rate Adaptation

Sensor hints aid in bit rate adaptation because node mobility affects wireless channel conditions, causing large and bursty changes over short intervals of time. When a node moves, bit errors and packet losses exhibit a higher degree of statistical correlation with past behavior as compared to the static case. We demonstrate this effect in Figures 6 (left) and 6 (middle).

Figure 6 (left) plots the *conditional probability* of losing packet number $i+k$ at a given bit rate, *given that* packet number i was lost, for different values of k (the "lag"). In this indoor experiment, we sent back-to-back 1000-byte packets at 54 Mbits/s from a stationary laptop to a stationary smartphone in the static case, and to a smartphone carried by a walking user in the mobile case. A link-layer ACK received from the smartphone indicated a packet success, otherwise the packet was considered lost. The graph shows a significantly higher loss probability for small values of k in the mobile case, demonstrating a larger degree of short-range dependence compared to the static case. In this scenario, for the mobile case, the next packet following a lost packet is significantly more likely to be lost than in the static case, and also compared to larger values of k. For both the static case and the mobile case, the unconditional loss probability was around 23%.

For the same traces, Figure 6 (middle) shows the mutual information between packet success/failure events separated by x ms. Specifically, we compute the mutual information between every pair of two success/failure events separated by a time interval of x ms for a range of different values of x. This measure shows the extent to which the fate of a later packet depends on the earlier one. In the static case, there is no mutual information between packets. But when a node moves, packets exhibit a higher degree of dependence with the past few packets. This dependence drops off at around 10 ms in these experiments. In Figure 6 (right), we plot the mutual information curve for different walking speeds and found the dependence to drop off at around 10–20ms.

These results show that the best strategy for bit rate adaptation is likely to be different when nodes move than when they are static. In more detail, in the static case, where the channel remains relatively stable, it makes sense to maintain a longer history of performance at different bit rates to smooth over periods of short-term fading or contention. Such a long-history approach falters when the device is mobile, because in the mobile case it makes more sense to keep only a short history, react quickly to errors, and perhaps sample other rates aggressively to track the faster changes typical of a mobile channel.

This observation motivates a *hint-aware bit rate adap-*

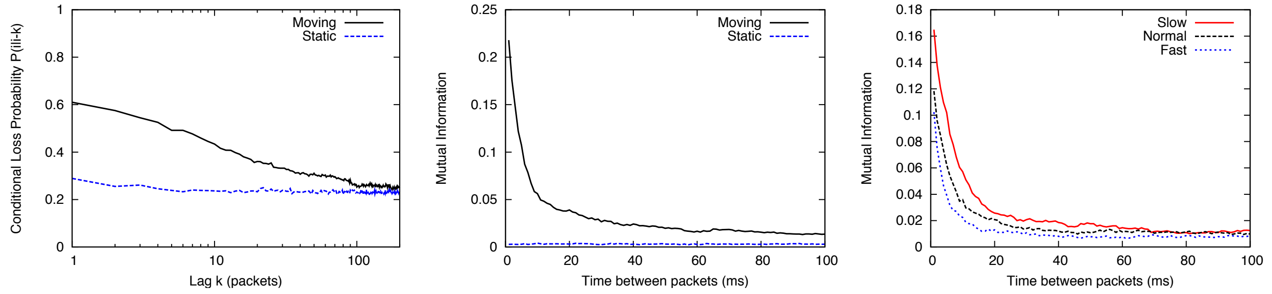


Figure 6: Left: Given a packet loss, the conditional probability of losing the kth packet following the loss, as a function of the lag, k. The unconditional loss probability for both the static and mobile cases was around 23%. Middle: Mutual information between packets separated by x ms specified by the x-axis value. In the static case, there is essentially no mutual information between packets, while in the mobile case, packets separated by less than 10 ms show a high degree of dependence. Right: Mutual information between packets separated by x ms for various walking speeds.

RapidSample$(lastbr, gotack)$:
  if $(!gotack)$ then
    $failedTime[lastbr] \leftarrow$ **CurrTime**$()$
    if $(sample)$ then
      $br \leftarrow oldbr$
    else
      $br \leftarrow max\{0, lastbr - 1\}$
      $sample \leftarrow 0$
  else
    $sample \leftarrow 0$
    if (**CurrTime**() - $pickedTime[lastbr] > \delta_{success}$) then
      $br \leftarrow max\{i \mid \forall j \leq i :$
          CurrTime$() - failedTime[j] > \delta_{fail}\}$
      $sample \leftarrow 1$
      $oldbr \leftarrow br$
    else $br \leftarrow lastbr$
  if $br \neq lastbr$
    $pickedTime[br] \leftarrow$ **CurrTime**$()$
  return br

Figure 7: The RapidSample bit rate adaptation algorithm. It is called for each packet with $lastbr$ describing the bit rate index and $gotack$ describing whether an ack was received for the previous packet. Time is reported in elapsed milliseconds.

tation scheme, which adapts differently depending on whether or not the nodes are moving. By using external sensor hints rather than making decisions based solely on network information, our goal is to combine schemes tuned separately for the static and mobile cases. The approach requires no training to achieve good performance.

With these remarks in mind, we introduce *RapidSample*, a frame-based rate adaptation protocol designed for a channel undergoing rapid changes due to movement.

3.1 The RapidSample Protocol

The RapidSample protocol is shown in Figure 7. It starts with the fastest bit rate. If a packet fails to get a link layer ACK, the protocol switches to the next lowest rate and records the time of the failure. After success at a particular bit rate for more than $\delta_{success}$ milliseconds (5 in our implementation), the sender attempts to sample a higher bit rate. It chooses the fastest bit rate: (a) that has not failed in the last δ_{fail} milliseconds (10 in our implementation), and (b) for which there is no slower bit rate that has failed within this interval. If the faster rate fails, it reverts to the original rate; if it succeeds, it adopts this new faster rate.

There are four ideas motivating RapidSample. First, we observed that when a packet fails while a node is moving, the probability of the next few packets failing at this bit rate is high (Figure 6, left). Therefore, the protocol immediately reduces the bit rate. Second, as we showed in our discussion of Figure 6 (middle), the mutual information between the the fate of packets x milliseconds apart becomes small when x is around 10–15 ms for all the indoor movement speeds we tested. We use a value of 10 ms for δ_{fail} as the minimum time to wait before sampling a previously failed rate, and before sampling any rate higher than the failed rate.

Third, RapidSample attempts higher rates after only a small number of successes at the current rate. We set $\delta_{success}$ to be less than δ_{fail}. In general, it is difficult to tell if the channel conditions are improving or degrading, but under movement, we posit that if conditions are not degrading, they are probably improving because it is unlikely that they are invariant. Thus, even a few successes at one rate provide enough confidence to sample higher rates that have not recently failed. Fourth, if we are wrong about the channel improving, and a higher rate fails, we immediately revert to the original rate.

3.2 Hint-Aware Bit Rate Adaptation Protocol

The Hint-Aware Rate Adaptation Protocol implemented at the sender uses RapidSample when a node is moving and uses SampleRate [3] when a node is static. It relies on movement hints from the receiver to switch between the two. We use SampleRate for the static case as it performed better than other frame-based and SNR-based protocols in various environments (see Section 3.3).

3.3 Evaluation

We use both trace-driven simulation and testbed experiments to evaluate our hint-aware rate adaptation scheme.

3.3.1 Trace-driven Simulation

To replicate the same mobility pattern between different experiments, we used trace-driven simulation—feeding real-world experimental data to a wireless simulator, allowing for both reproducibility and realism. We used the same experimental architecture as [25], which modified the ns-3 network simulator (*v*3.2) to read in experimental traces describing, for each 5 ms time slot, the fate of each packet sent at each bit rate during that time slot. This setup bypasses the physical layer's propagation model, instead referencing the trace file to determine if a packet should be received successfully.

To collect the traces, we configured a Linux laptop as a sender. It ran the Click router using the MadWiFi 802.11 driver, which in turn used an Atheros 802.11 chipset. The laptop sent a constant stream of 1000 byte packets, cycling through the 802.11a OFDM bit rates of 6, 9, 12, 18, 24, 36, 48, and 54, in round-robin order. Each cycle through all 8 bit rates took approximately 5 ms. Indoors, we used 802.11a to minimize interference with local infrastructure networks. We configured a second laptop with the same hardware to act as a receiver, logging every received packet. This laptop was additionally equipped with a SparkFun serial accelerometer for movement hints.

We collected several traces from four different environments for static and mobile scenarios: 1) an *office setting* with no line-of-sight between the sender and receiver, 2) a *long hallway* with line-of-sight between the nodes, 3) an *outdoor setting* with a lightly crowded outdoor pavement area, and 4) a *vehicular setting* where the sender is stationary on the roadside and the receiver is in a moving car near MIT (an urban area).

We evaluated the following frame-based bit rate adaptation protocols: RapidSample, SampleRate [3], RRAA [26], and our *hint-aware* method that switches between RapidSample and SampleRate, depending on the sensor hint. We also evaluated two SNR-based rate adaptation protocols: RBAR [7] and CHARM [8]. For both these schemes, we trained the protocol for the operating environment. We also assumed that the sender has up-to-date knowledge about the receiver SNR. Finally, we compared our protocol to SoftRate [25], a bit rate adaptation scheme that uses SoftPHY hints from a modified physical layer and which can adapt the bit rate on a per-packet basis without requiring training. For this comparison we used the traces from [25].

Figure 8 shows the performance of the hint-aware protocol compared to the other rate adaptation protocols for three of the four environments (we discuss the vehicular setting later in this section). For each environment, we collected 10–20 traces. Each trace is 20 seconds long with 50% static and mobile periods. The receiver was static for 10 seconds and mobile for 10 seconds in each trace. The workload we used was TCP. The graph shows the average TCP throughput of all the schemes as a fraction of the throughput obtained by the hint-aware protocol. The error bars show the 95% confidence interval. In every environment, the hint-aware protocol obtained significant performance gains. It improved over SampleRate by 23% to 52% on average, over RRAA by 17% to 39%, and over RBAR by 11% to 47%. We do not show the numbers for CHARM as the performance of RBAR and CHARM was similar in all cases with RBAR performing slightly better.

We also evaluated the different protocols separately for mobile and static scenarios. For each scenario, we collected ten 20-second traces in each of the test environments. Figure 9 shows the average TCP bulk transfer throughput of all the schemes as a fraction of the throughput obtained by RapidSample, in the mobile case. RapidSample performed significantly better than other schemes in every environment. It obtained up to 75% better throughput on average than SampleRate and up to 25% better than other protocols. It achieved about 28% more throughput than SampleRate, 36% more throughput than RRAA and nearly 2× more throughput than the SNR-based protocols. These performance gains come from RapidSample's ability to cope up with the rapid fluctuations in the channel conditions when a node is mobile.

On the other hand, RapidSample is the worst-performing protocol in the static case, as shown in Figure 10. It achieved 12% to 28% *lower* average throughput compared to SampleRate and up to 18% *lower* throughput compared to RRAA. The poor performance is because RapidSample aggressively reduces the rate even on a single loss and frequently tries to sample higher rates even when the channel conditions are not changing. Figure 10 also shows that SampleRate usually achieved higher throughput than other protocols when the nodes are static. Hence, we decided to use SampleRate for the static case in our hint-aware rate adaptation protocol.

We also measured the performance of RapidSample in a vehicular setting, where the sender was stationary on the roadside and the receiver was placed in a moving car.

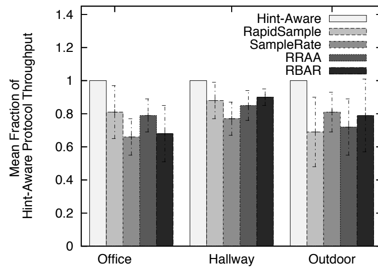


Figure 8: Hint-aware protocol performs better in mixed-mobility setting.

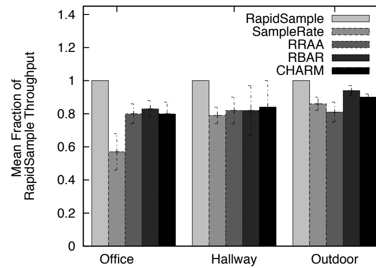


Figure 9: In mobile scenarios, RapidSample performs significantly better than other protocols.

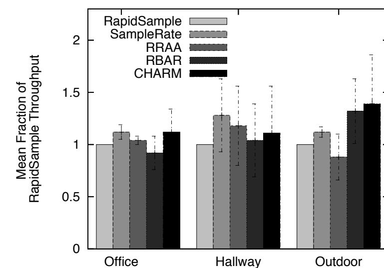


Figure 10: In the static case, RapidSample performs poorly compared to the other schemes.

We collected 10 traces, each 10 seconds long. Figure 11 shows the results, where the traffic workload is UDP (at a rate of 36 Mbps), as TCP repeatedly times out when faced with high packet loss rate [6]. Similar to other mobile environments, RapidSample outperformed the other schemes.

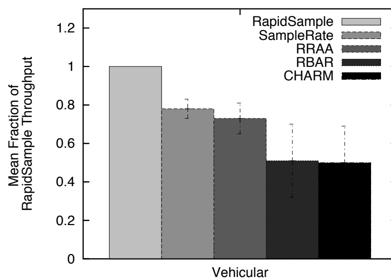


Figure 11: In vehicular environments, RapidSample achieves significantly higher throughput compared to other schemes.

Finally, in Figure 12, we compare RapidSample to SoftRate, SampleRate, and RRAA, using the walking traces and ns-3 protocol implementations from [25]. RapidSample performs nearly as well as SoftRate on these traces, without the aid of SoftPHY hints, further confirming the effectiveness of RapidSample in mobile settings. As a result, our hint-aware protocol performs nearly as well as SoftRate, but is readily deployable on many of today's commodity devices.

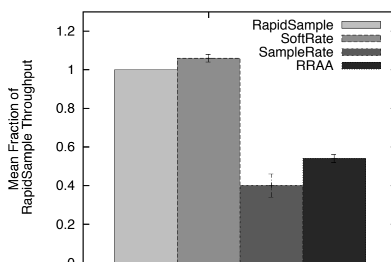


Figure 12: RapidSample performs almost as well as SoftRate on traces collected while walking.

3.3.2 Testbed Experiments

Trace-driven evaluation allows us to compare the performance of various protocols, but there might be a concern that the method used does not correctly account for the channel variations observed in practice. To show that the scheme can work in real-time, we deployed a testbed of Android Nexus One smartphone receivers communicating with a MadWiFi-based Linux laptop sender. We implemented the frame-based rate adaptation protocols (SampleRate, RRAA, RapidSample, and our hint-aware protocol) on the laptop as user-level Click modules; we did not implement the SNR-based protocols as they required SNR feedback from the receiver. The hint-aware protocol used the Sensor Hint Service on the laptop to monitor for movement hints from the smartphone. It switched between SampleRate and RapidSample schemes based on movement hints. The implementation on the smartphone instructed the Sensor Hint Service to send movement hints to the laptop using UDP. The movement hints were sent every second and on hint changes ("static to moving" or "moving to static").

We configured the laptop to send 802.11 data packets to a smartphone's wireless MAC address. Upon receiving the packet, the phone responds with a link-layer ACK. We put the phone in tethering mode, to disable the 802.11 power-saving mode that was on by default. We measure the performance of bit rate adaptation based on the received ACKs.

We evaluated the protocols in two environments: an *office setting* and a *long hallway* setting, the same as in the trace-based evaluation. In each environment, we used 10 distinct mixed-mobility patterns and measured the throughput of each scheme. In each mobility pattern, a user walked in a predefined trajectory at a constant speed and stayed static at predefined locations for predefined amounts of time. Each such *trial* took 45–90 seconds to complete and had an equal amount of static and walking periods. The phone was held by the user in the same way across experiments. Since it was hard to exactly replicate the same mobility pattern, we repeated each trial 3 times

and report the average and the standard deviation. A trial consists of running SampleRate, RRAA, RapidSample and the hint-aware protocol back-to-back for the same mobility pattern. Three back-to-back trials correspond to one experimental *run*.

Because the smartphone only had a 802.11b/g card, we did all these experiments in the relatively busy 802.11b/g channels. To minimize interference from the existing access points, we ran the experiments late at night. In every experiment, we sent a stream of 1000-byte packets back-to-back. The bit rate of each outgoing packet was controlled by the rate adaptation scheme. We measured the throughput based on the received link-layer ACKs. The maximum throughput we were able to obtain from the user-level Click implementation was around 27 Mbps.

Figure 13 (left) shows the measured throughput of different protocols in the two environments. For each environment, we plot the average throughput and standard deviation (as error-bars) for 10 different runs. The charts show the results sorted by the throughput of the hint-aware scheme.

In both environments, the hint-aware protocol consistently outperforms the other schemes. In the office setting, it improved over SampleRate by between 10% and 76%, over RRAA by between 8% and 100%, and over RapidSample by between 11% to 41%. On average, it obtained 20% more throughput than SampleRate, 22% more throughput than RRAA, and 19% more throughput than RapidSample. In the hallway setting, the hint-aware protocol obtained 9%–49% more throughput than SampleRate, 8%–80% more throughput than RRAA, and 5%–85% more throughput than RapidSample. On average, it improved over SampleRate, RRAA, and RapidSample by 17%, 37%, and 22% respectively.

Compared to trace-driven results, SampleRate performed better than RRAA in these testbed experiments, especially in situations where the throughput of all the schemes was low. RRAA performed better when the throughput was higher. Otherwise, the testbed results were fairly consistent with the trace-driven simulations.

During each trial, for every packet sent, in addition to logging if an ACK was successfully received, we logged the movement hint as well. We processed these traces and used the movement hint to split them into static and mobile phases and measured the throughput separately for each scenario. Figure 13 (middle) shows the average throughput obtained during the mobile phases of the corresponding experimental runs shown in Figure 13 (left). In mobile scenarios, RapidSample performs significantly better than SampleRate and RRAA in both the environments. On average, it improved over SampleRate by 61% and 40% in the two environments and over RRAA by 16% and 39%. The relative performance of SampleRate was worse in the office setting compared to the hallway setting. This result is consistent with what we found in the trace-driven evaluation. Similarly, Figure 13 (right) plots the mean throughput for the static phases. As found in the trace-based simulation, SampleRate is the best protocol in the static case and RapidSample performed much worse than SampleRate and RRAA.

3.4 Discussion

In our scheme we use only a binary movement hint that indicates whether the device is stationary or mobile. An important conclusion from our results is that even such a simple hint can produce significant performance improvements. An obvious future direction is to generalize our scheme to use additional hints such as speed and heading. Using Figure 6 (right), it is possible to fine-tune parameters in RapidSample for different speeds. While it is easy to get a movement hint, measuring speed accurately indoors using the sensors available on a smartphone is a challenging unresolved problem.

The use of hints for bit rate adaptation may improve PHY-assisted techniques such as SoftRate [25], which perform significantly better than existing protocols in the mobile case using an instantaneous estimate of the bit error rate. Augmented with a movement hint, however, they may be able to adapt their behavior in the static case to average these estimates and avoid reacting to short-term fading.

One benefit of using the accelerometer for a movement hint is that it detects even small movements of the device—e.g., a user moving a smartphone from his head to pocket—which can change the channel conditions. Of course, it is also possible that the channel conditions can change due to changes in the surrounding environment, even if the device is stationary. If such changes are short-lived, then SampleRate, the protocol we use during stationary periods, performs well.

Our trace-driven evaluation shows that the hint-aware protocol performs better than trained SNR-based adaptation in all the tested environments. One question that might arise is whether a protocol could simply use information about *changes* in the observed RSSI values to infer movement and use a protocol like RapidSample in that case, without relying on external sensor hints. We explored this approach and found several problems with it. First, RSSI values showed large variations even when a node was static. Second, the magnitude of these variations depended strongly on the environment and the device. It also varied significantly across time and across different RSSI ranges (low RSSI ranges showed more fluctuations than high RSSI ranges). Third, the reported RSSI was extremely sensitive to movement in the environment and triggered many false hints. Hence, using RSSI was more error-prone than using explicit hints. Furthermore explicit hints avoid the need for training. It

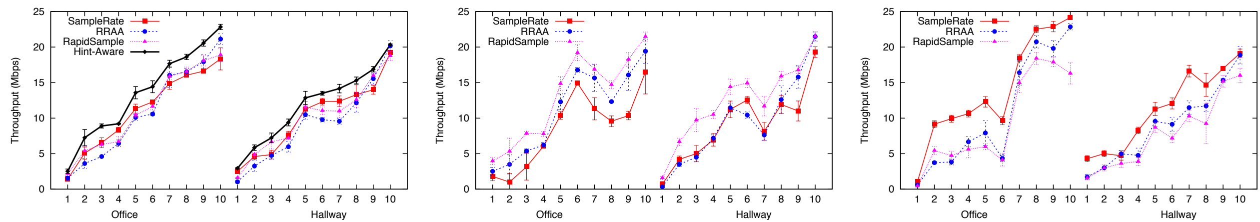


Figure 13: Throughput of the different bit rate adaptation protocols. The left-most chart shows all the protocols with one data point per run; the error-bars are the standard deviations. There are ten runs inside offices and ten in the hallways, with each run lasting 45–90 seconds. The middle chart shows the results considering only times when the device was moving, while the right-most chart shows the results from the same runs considering only times when the device was static. In these experiments, the hint-aware protocol was better than the next-best protocol SampleRate by between 20% (office) and 17% (hallway), with a mean overall improvement of 19%. When mobile, RapidSample outperformed SampleRate by 61% (office) and 40% (hallway), with a mean overall improvement of 50%.

is, of course, conceivable that one could combine RSSI and sensor hints to further improve bit rate adaptation; achieving this goal without environment-specific training remains an open question.

4 Hint Aware AP Association

Most wireless clients associate with the AP with the strongest RSSI (SNR) value. When the RSSI falls below some fixed threshold, the client triggers a handoff, where it scans all the channels and associates with the AP with the strongest RSSI. We refer to this approach as the *standard scheme*.[1] As others have observed [14, 18], the standard scheme is sub-optimal in many settings, particularly when the client is mobile. In this section, we develop a hint-aware association protocol that performs better than the standard scheme.

In our scheme, a node detects whether it is indoors or outdoors using a GPS lock hint. If it is indoors, its association strategy uses the "walking" hint (Section 2.2) to detect motion. The protocol may be configured at run-time to either maximize throughput (Section 4.1), or minimize the number of handoffs (Section 4.2); the former is useful for bulk transfers, while the latter is useful for interactive real-time applications such as telephony for which the hundreds of milliseconds spent during a handoff will disrupt communication, increasing both jitter and packet loss [9] (handoffs took approximately 600 ms on the Android smartphones used in our experiments). When a node is outdoors, it implements a similar strategy, using the position and speed as hints. We do not evaluate the outdoor case in this paper.

We implemented our association protocol as an easily deployable background Android application. Below, we describe the two modes of the protocol and evaluate their performance. Our experimental results with indoor mobility show a median throughput increase of 30% and a median reduction of 40% in the number of handoffs compared to the standard scheme.

[1] Some association schemes do include periodic scans, but they are done only every few minutes, and never while transferring data.

4.1 Using Hints to Maximize Throughput

We present a hint-aware AP association strategy for maximizing throughput. The strategy builds on three observations. First, when a client is moving, the probability that a new AP with a stronger signal enters its range is higher than when the client is static. Hence, when mobile, a client should scan periodically to discover better APs: the throughput gain of associating with a better AP is likely to be higher than the throughput lost to the scan. The periodicity depends on the speed of the client and the expected range of the typical AP in the deployment.

Second, when a client is stationary, it is less likely that new, better, APs will be discovered. In this case, the penalty of a scan is not worth incurring.

Third, when a client stops moving, it may remain static for some period of time. If so, it is worth performing a scan on this transition because the AP with the strongest RSSI is likely (though not guaranteed) to remain optimal until the client moves again. When static, a client should re-scan and re-associate only when it starts moving again, or when the current AP's RSSI becomes weaker than some threshold. In our experiments with the standard scheme, when a client moves from one location to another nearby location, in many cases it remains associated with the original AP (because the signal strength remains above the handoff threshold), reducing throughput. By rescanning immediately following a transition from mobile to static, we avoid this problem.

Our protocol is simple. When the association daemon running on the client detects that the client is walking, it scans periodically, every T_{sc} seconds, for the AP with the highest RSSI. If the client goes from the moving to static state, it performs a scan immediately and associates with the strongest AP. When it is static, it performs no scans, unless the RSSI drops below a threshold or if the client starts moving.

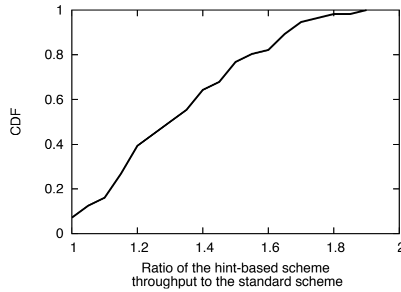


Figure 14: CDF of the ratio of throughput obtained by the hint-aware scheme to the throughput obtained by the standard scheme, calculated for 30 traces.

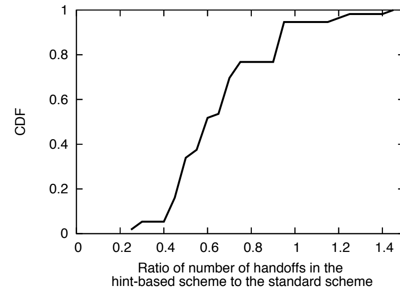


Figure 15: CDF of the ratio of number of handoffs using the heading-aware handoff scheme to the standard scheme calculated over the 30 testing tracks.

The ideal value of T_{sc} is the time it is likely to take for the current AP to no longer be the best one while the user is moving—a factor which depends on how APs are deployed and how fast the user is moving. To get a sense for what it might be in practice, we wrote a data collection application on the Android platform that scans every second, recording the signal strength of every AP it hears. It also records the walking and heading hints with each scan. We convert each RSSI value to a throughput value using a rate map as in [11].

We collected several such traces with different movement patterns and pedestrian speeds indoor in two different buildings on the MIT campus. We found that at pedestrian speeds, a value of $T_{sc} = 8$ seconds maximized throughput. In other words, 8 seconds is approximately the time required to walk between two adjacent APs.

Performance evaluation. To quantify the throughput gains of our hint-aware protocol, we collected 30 walking traces in MIT hallways. These traces are different from the ones we analyzed to determine the value of T_{sc}, but the setting was the same. We had the client transition from moving to stationary states randomly, with roughly equal time spent in each state.

We performed a trace-based evaluation of our hint-aware association protocol compared to the standard scheme, on these traces. Figure 14 shows the CDF of the ratio of throughput obtained by our scheme to the throughput obtained by the standard scheme. The median throughput improvement is about 30%.

4.2 Using Hints to Minimize Handoffs

We now present a hint-aware AP association strategy for minimizing the number of handoffs, which is useful for applications such as VoIP. Our protocol requires lightweight training that can be deployed as a background application on standard phones. The protocol uses the observation that to minimize handoffs, the AP with the strongest RSSI is not necessarily the AP that will yield the longest-lasting connection. If a client is moving towards an AP, for example, it is likely to stay connected longer than if it is moving away, even if the RSSI at the time of the scan is not the highest among the set of observed APs. Our protocol uses heading hints to aid such decisions.

To train our protocol for a specific environment, we use the Android data collection application described earlier. Every second, this application logs the device's heading along with a list of APs and their signal strengths. We use this data to compute a model that maps from a <heading, current AP> pair to a preferred AP, where the preferred AP is the AP to associate with when handing off from the current AP at the given heading to maximize connection time.

Once trained, the protocol works as follows. If it detects the client is stationary it uses the standard scheme. If the protocol detects the client is walking, it extracts a heading hint. It then queries the model using this heading hint and its currently associated AP. The model looks up similar <heading, AP> pairs, and returns the AP that the client should associate with once the current AP's signal strength drops below the handoff threshold. The model attempts to select the AP that will maximize connection time. If the AP returned by the model is not seen during the scan for handoff, the protocol defaults to the standard method of choosing the AP with the highest RSSI.

To evaluate our protocol, we collected 60 tracks using several Android phones, walking through various MIT hallways. For each track, the user chose an arbitrary path in the building complex, and walked between 3–5 minutes. We split the data into training and testing sets—training using the former and testing using the latter. Figure 15 shows that the number of handoffs in our scheme is 40% lower than in the standard scheme. It also shows that for over 90% of the traces, our protocol yielded an improvement of at least 10%.

5 Topology Maintenance

In this section, we study the use of hints to improve the accuracy and efficiency of topology maintenance in wireless mesh (and sensor) networks. Here, each node often maintains a list of neighbor nodes along with the quality of connectivity to each neighbor. The standard method

for maintaining neighbors and link quality information in this setting is for a node to send periodic probe packets. Each recipient maintains the packet loss rate of packets from its neighbor, and may exchange this information in the routing protocol's messages. A key parameter is the *probing rate*: how often should these periodic messages be sent? In practice, a node may send these messages at more than one bit rate to produce link quality information at different bit rates.

In determining the frequency of these probes, two opposite considerations must be reconciled. On the one hand, sending frequent probes allows the nodes to maintain an accurate estimate of link qualities and identify changing topologies. Maintaining accurate values for this metric is important to avoid packet losses, which can increase the number of retries and also incorrectly slow down the bit rate. On the other hand, frequent probe packets themselves use large chunks of the bandwidth and increase network contention. This tradeoff becomes even more acute in mobile settings, where link quality changes rapidly. For instance, Figure 16 (left) captures the channel behavior that we observed in a mixed stationary/mobile setting. This plot shows the packet delivery ratio when the user is moving (derived from our movement hint) over time for one specific trace. To calculate the delivery ratio, we bucketed time into intervals of 1 second, during which time the sender transmits approximately 200 packets at each bit rate. The key observation is that motion causes the packet delivery ratio to fluctuate, with many of the jumps in the delivery ratio exceeding 20%.

Our idea is simple: because channel conditions vary much more in the presence of movement, probe frequently when a node receives movement hints from its neighbor or itself moves, and probe less often when the nodes are static.

5.1 Measurement

To evaluate the potential gains, we gathered data in an indoor environment where the sender was static and the receiver was either at a fixed location (*stationary*) or was moved at walking speeds (*mobile*). The sender sends probes at a rate of 200 probes per second. We calculate the actual delivery probability over a sliding window of 10 packets from these rapidly sent probes, subsampling the outcome of these probes to determine the delivery probability at various lower probing rates. We collected 20 stationary and 20 mobile traces, each 180 seconds long. We aggregate the results of the static cases into one set, and the mobile cases into another set. For each set, we calculate the *error in the delivery probability estimate*, which depends on the probing rate, as |Observed probability − Actual probability|.

Figure 16 (middle) shows the *average* error in delivery probability calculated from all the error samples for the static case as a function of the probing rate; the error bars show the standard deviations. Even a low probing rate of 1 packet every 10 seconds has an error of only 11%, suggesting that the default probing rate of many wireless networks of 1 probe/s may be too high. In contrast, Figure 16 (right) shows that the error in delivery probability is much higher in the mobile case, exceeding 35% even at a probing rate of 1 packet every 2 seconds. To achieve an error of about 10%, the mobile case requires a probing rate of 5 probes per second, which is more than 25× higher than for the static case at the same error rate. For a desired error of 5%, the mobile case needs 10 probes/s, while the corresponding rate for the static case is 0.5 probes per second, a 20× difference.

To understand the reason for this difference, consider a representative 25-second mobile trace in Figure 17 (left). The estimated probability does not track the actual probability except at a high probe rate. This differs from what is observed in the static case.

5.2 Hint-Aware Topology Maintenance Protocol

We implemented a hint-driven topology maintenance protocol using rates of 1 and 10 probes per second for the stationary and mobile cases, respectively. The protocol continues to send at the fast probe rate for one second after the node stops moving in order to estimate the correct metric, before slowing the probe rate down.

Figure 17 (right) compares the performance of our protocol to the standard 1 probe/s protocol. We also plot the movement hint, with a raised value indicating movement. Notice that our adaptive protocol maintains an accurate assessment of the actual delivery probability throughout the experiment, while the non-adaptive 1 probe/s strategy lags by multiple seconds. Note that in some cases, the 1 probe/s approach mis-estimates the delivery probability by more than 30%, whereas, the adaptive estimator is always within 5%.

A simple analysis shows how link mis-estimation degrades throughput. Suppose a node uses ETX [5] to pick the next-hop. Suppose further that there are two choices, one with link delivery probability p_1 and the other with probability p_2; without loss of generality, let $p_1 > p_2$. ETX would choose link 1, with metric $1/p_1$.

Let the error in the average link delivery probability estimate be δ (Figure 16 (right) shows that $\delta > 0.25$ in some cases). The node would pick the wrong link if $p_2 + \delta > p_1 - \delta$. In this case, the extra number of transmissions relative to the optimal value is $\frac{1}{p_2} - \frac{1}{p_1}$. The overhead relative to the optimal is $\frac{p_1}{p_2} - 1$, which can be large for practical parameters; e.g., if $p_1 = 0.8$ and $p_2 = 0.6$, the throughput reduction is 33%.

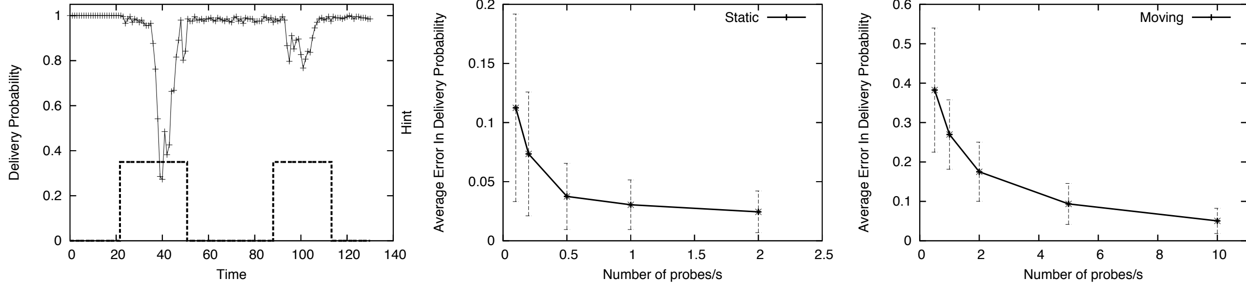


Figure 16: Left: Packet delivery rate for 6 Mbps packets over time; the raised dashed hint line indicates the device is moving. Middle & Right: Average error in delivery probability versus probing rate for static and mobile cases.

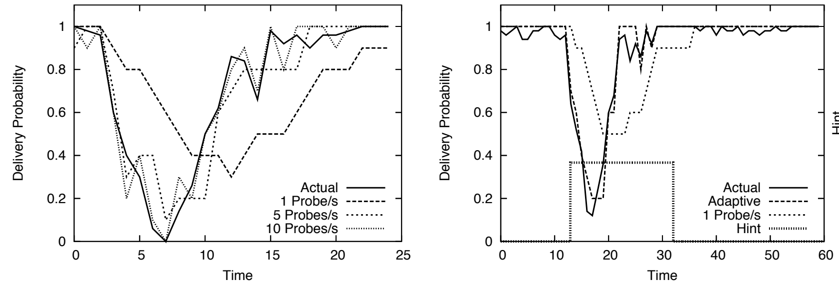


Figure 17: Delivery probability over time for the mobile trace (left) and for a combined static and mobile trace with the dots showing the movement hint (right).

6 Vehicular Network Path Selection

We now investigate whether hints can improve path selection in vehicular mesh networks. Networking strategies in this setting are complicated by dynamic neighbor tables, which can generate a high rate of broken paths. In general, broken paths increase overhead and latency. For this reason, selecting paths with the longest expected connection time may be a good idea.

6.1 Connection Time Estimate Metrics

We present three connection time estimate (CTE) metrics that use *heading* and *speed* hints to estimate whether a path in a vehicular network is likely to be long-lived. Let the ordered sequence $(u_1,...,u_j)$ denote a $j-1$ hop path, $dh(u_i,u_{i+1})$ denote the difference in heading of nodes u_i and u_{i+1} (from 0 to 180 degrees), and $s(u_i)$ denote the speed of u_i (in m/s). Our three CTE metrics, called $cte1$, $cte2$, and $cte3$, are defined for a path $R=(u_1,...,u_j)$:

$$\begin{aligned} cte1(R) &= \prod_{u_i,u_{i+1}\in R} \frac{1}{dh(u_i,u_{i+1})} \\ cte2(R) &= \min_{u_i,u_{i+1}\in R} \left(\frac{1}{dh(u_i,u_{i+1})}\right) \\ cte3(R) &= cte1(R)\cdot \frac{1}{1+\sum_{u_i\in R} s(u_i)} \end{aligned}$$

The metrics use the assumption that a small difference in heading indicates nodes are moving in the same direction on the same road, and are therefore likely to stay connected longer. The $cte3$ metric includes the additional assumption that speed is inversely correlated with connection duration. Because $cte1$ multiplies the inverse of heading differences at each hop, it is biased toward single-hop paths. The $cte2$ metric, by contrast, evaluates a path only by its worst hop, scoring multi-hop paths higher than $cte1$. The $cte3$ metric multiples the inverse of the sum of node speeds with the $cte1$ value. It follows, for example, that doubling the speed of each node on a path approximately halves its $cte3$ score.

To calculate these metrics, each node appends its heading and speed to its mesh neighbor probes. For all three, larger values predict longer-lived paths. These metrics are simple, and require no knowledge of road geometry.

6.2 Evaluation

We evaluated these metrics over a collection of vehicular mobility traces generated from raw position samples gathered from vehicles in the CarTel project over the duration of a year in the Boston metro area, map-matched to an underlying road network [23]. We combine a collection of traces into a *network*, and then simulate, for each second, the position of every vehicle in the network, adjusting the traces so they all appear to begin at the same time. We consider two vehicles to have a *link* at a given time if and only if they are within 100 meters at that time in their traces (we use geographic proximity as a crude surrogate for connectivity).

We measured the relationship between CTE values and path duration over both one and two hop paths.

Specifically, we studied 15 networks consisting of 100 vehicles each. Each simulation lasted for 120 seconds. For each of the over $190,000$ routes observed in these simulations, we calculated all three CTE metrics when the path is first formed, and the total duration of the path (in seconds) before it breaks. For each metric, we bucket the CTE scores (into buckets of size $1/20$ for $cte1$, $1/10$ for $cte2$, and $1/200$ for $cte3$), and calculate the median link duration of the paths in each bucket. In Figure 18 we plot these durations for the first three buckets (in descending order of associated CTE score) for each CTE metric. The dashed line indicates the median duration over all paths.

The figure shows that all three CTE metrics provide an effective *filter* for long-lived paths. If a path's $cte1$ value falls into the first bucket, or if its $cte2$ or $cte3$ values fall into the first two buckets, then the path is likely to be long-lived. The median duration of paths in these buckets is 2–5$\times$ longer than the median over all paths.

Identifying long-lived paths might not be a good strategy if the selection mechanism is somehow biased toward routes with low throughput. To evaluate this possibility, we use distance as a rough approximation of achievable throughput (we only have position data from the networks used in this evaluation). We plot in Figure 19 the CDF of time versus distance for the single-hop paths in the first bucket of $cte1$, and the first two buckets of $cte2$, and $cte3$. For comparison we also plot the function for all single-hop paths. This figure confirms that our CTE metrics show no bias favoring links of larger distances (lower throughput).

7 Limitations

Energy. Sampling sensors consumes energy and reduces the battery lifetime of a mobile device. Figure 20 shows the battery lifetime of an Android G1 device when various hints are sampled at the highest supported rates. Notice that the accelerometer and compass consume much less energy than GPS. To alleviate energy concerns, protocols should extract hints only when transferring data. Moreover, sensors like the accelerometer on a mobile device are usually always on by default (for instance, to continuously detect changes in screen orientation), so extracting hints from them should consume no extra energy. *Triggered sensing* [10] can further reduce the energy consumed by some sensors. Here, a low-power sensor turns on or off a high-power sensor based on certain events; for example, GPS can be turned on only when the accelerometer detects movement. We can also dynamically reduce the sampling rate of sensors to reduce the energy cost [22, 23, 24], and replace expensive GPS with lower-energy position sensors like GSM radios, as in CTrack [24]. In addition, sensor hints can be turned off when the battery is low and protocols can revert to a hint-unaware scheme.

Calibration across devices. The disparity between sensors across different devices and platforms might pose a challenge for hint-aware protocols to work without sensor calibration and tuning. We have implemented the Sensor Library for Android Nexus One, Android G1, and iPhones. The movement hint worked seamlessly across these platforms, but the walking hint detector [22] required a little tuning for each *type* of device.

Privacy. Sharing mobility hints with other nodes might expose private information. For instance, by continuously monitoring movement and heading hints, it might be possible to track a user's behavior more accurately than by just monitoring wireless packets from a device (e.g., I might be able to determine more reliably that you left your office because of the movement hints broadcast by your device); one might alleviate this problem by having all communication go via a (trusted) AP, and encrypting the hints sent to the AP.

8 Related Work

To the best of our knowledge, ours is the first practical work to explore the benefits of systematically integrating sensor hints into a wireless network architecture. Related work that uses information outside the wireless networking stack has mostly focused on wireless power saving. For instance, Wake on Wireless [17] uses an additional low power radio that can be used for signaling to wake up the wireless radio. Cell2Notify [1] uses the cellular radio on a smartphone to wakeup the WiFi interface for VoIP calls thus reducing the energy consumption of WiFi. BlueFi [2] uses GSM towers and nearby Bluetooth devices to predict if WiFi connectivity is available, hence achieving power savings.

In addition to power savings, hints from external sensors for wireless protocols have been used, usually in vehicular network designs. Mobisteer [12] uses directional antennas in vehicles and location hints from GPS to find the best antenna orientation and the AP to associate with. Breadcrumbs [13] predicts the best AP to associate with using a mobility model built using GPS coordinates. CARS [16] is an inter-vehicle bit rate adaptation protocol that uses knowledge of the speed and distance between communicating cars to pick a bit rate. Their method collects a large amount of training data for an environment to determine the best bit rate to use at different speeds and distances; in contrast our hint-aware bit rate adaptation method does not require any such training and performs well across a variety of conditions.

9 Conclusion

This paper introduced a network architecture that uses sensor hints to augment and improve wireless protocols.

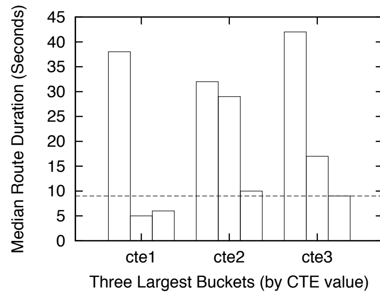


Figure 18: The median route duration for the highest three CTE value buckets. The dashed line is the median over all routes.

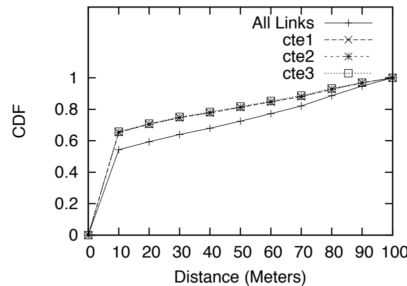


Figure 19: CDF of time spent at each distance for all one-hop links, and the one-hop links from the first $cte1$ bucket, and the first two $cte2$ and $cte3$ buckets.

Hint Type	Sensor	Approximate Lifetime (hr)
Movement	Accel	19
Walking	Accel	19
Heading	Compass	18
Heading	GPS	6
Speed	GPS	6
Environment	GPS	6
No hint extraction		22

Figure 20: Battery lifetime of Android G1 for continuous hint monitoring (with screen at minimum brightness).

The key idea is to use these hints to infer the context in which communication is occurring, and to use that context to adapt the behavior of protocols. We applied this idea to develop hint-aware protocols for bit rate adaptation, access point association, topology maintenance, and path selection in vehicular networks. Sensor hints can also augment other protocols, as described in our earlier position paper [15]. These include: adapting the length of the cyclic prefix to outdoor speeds, scheduling client traffic at an AP taking movement into account, preemptively disassociating clients that have likely moved beyond the range of an AP, and network monitoring.

ACKNOWLEDGMENTS

We thank Aditya Akella for several useful comments. This work was supported by the National Science Foundation under grants CNS-0931550 and CNS-0721702.

REFERENCES

[1] 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*, 2007.

[2] G. Ananthanarayanan and I. Stoica. Blue-Fi: Enhancing Wi-Fi Performance Using Bluetooth Signals. In *MobiSys*, 2009.

[3] J. Bicket. Bit-rate Selection in Wireless Networks. Master's thesis, Massachusetts Institute of Technology, February 2005.

[4] J. Camp and E. Knightly. Modulation Rate Adaptation in Urban and Vehicular Environments: Cross-layer Implementation and Experimental Evaluation. In *MobiCom*, 2008.

[5] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A High-throughput Path Metric for Multi-hop Wireless Routing. In *MobiCom*, 2003.

[6] J. Eriksson, H. Balakrishnan, and S. Madden. Cabernet: Vehicular Content Delivery Using WiFi. In *MobiCom*, 2008.

[7] G. Holland, N. Vaidya, and P. Bahl. A Rate-adaptive MAC Protocol for Multi-Hop Wireless Networks. In *MobiCom*, 2001.

[8] G. Judd, X. Wang, and P. Steenkiste. Efficient Channel-aware Rate Adaptation in Dynamic Environments. In *MobiSys*, 2008.

[9] A. Mishra, M. Shin, and W. Arbaugh. An Empirical Analysis of the IEEE 802.11 MAC Layer Handoff Process. *SIGCOMM CCR*, April 2003.

[10] P. Mohan, V. N. Padmanabhan, and R. Ramjee. Nericell: Rich Monitoring of Road and Traffic Conditions using Mobile Smartphones. In *SenSys*, 2008.

[11] R. Murty, J. Padhye, R. Chandra, A. Wolman, and B. Zill. Designing High Performance Enterprise Wi-Fi Networks. In *NSDI*, 2008.

[12] V. Navda, A. Subramanian, K. Dhanasekaran, A. Timm-Giel, and S. Das. MobiSteer: Using Directional Antenna Beam Steering to Improve Performance of Vehicular Internet Access. In *MobiSys*, 2007.

[13] A. J. Nicholson and B. D. Noble. BreadCrumbs: Forecasting Mobile Connectivity. In *MobiCom*, 2008.

[14] I. Ramani and S. Savage. SyncScan: Practical Fast Handoff for 802.11 Infrastructure Networks. In *Infocom*, 2005.

[15] L. Ravindranath, C. Newport, H. Balakrishnan, and S. Madden. "Extra-Sensory Perception" for Wireless Networks. In *HotNets*, 2010.

[16] P. Shankar, T. Nadeem, J. Rosca, and L. Iftode. CARS: Context Aware Rate Selection for Vehicular Networks. In *ICNP*, 2008.

[17] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *MobiCom*, 2002.

[18] M. Shin, A. Mishra, and W. Arbaugh. Improving the Latency of 802.11 Hand-offs using Neighbor Graphs. In *MobiSys*, 2004.

[19] Smartphone Owners Lead Rise in Mobile Internet Usage. https://www.strategyanalytics.com/default.aspx?mod=ReportAbstractViewer&a0=5100.

[20] More Smartphones Than Desktop PCs by 2011. http://www.pcworld.com/article/171380/more_smartphones_than_desktop%25%20_pcs_by_2011.html.

[21] S. H. Stovall. *Basic Inertial Navigation*. Naval Air Warfare Center Weapons Division, 1997.

[22] A. Thiagarajan, J. P. Biagioni, T. Gerlich, and J. Eriksson. Cooperative Transit Tracking Using GPS-enabled Smart-phones. In *SenSys*, 2010.

[23] A. Thiagarajan, L. Ravindranath, K. LaCurts, S. Toledo, J. Eriksson, S. Madden, and H. Balakrishnan. VTrack: Accurate, Energy-Aware Traffic Delay Estimation Using Mobile Phones. In *SenSys*, 2009.

[24] A. Thiagaran, L. Ravindranath, S. Madden, H. Balakrishnan, and L. Girod. Accurate Low Energy Map Matching For Mobile Devices. In *NSDI*, 2011.

[25] M. Vutukuru, H. Balakrishnan, and K. Jamieson. Cross-layer Wireless Bit Rate Adaptation. In *Sigcomm*, 2009.

[26] S. Wong, H. Yang, S. Lu, and V. Bharghavan. Robust Rate Adaptation for 802.11 Wireless Networks. In *MobiCom*, 2006.

[27] Smartphone Sales Up 24 Percent. http://techcrunch.com/2010/02/23/smartphone-iphone-sales-2009-gartner/.

Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center

Benjamin Hindman, Andy Konwinski, Matei Zaharia,
Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

We present Mesos, a platform for sharing commodity clusters between multiple diverse cluster computing frameworks, such as Hadoop and MPI. Sharing improves cluster utilization and avoids per-framework data replication. Mesos shares resources in a fine-grained manner, allowing frameworks to achieve data locality by taking turns reading data stored on each machine. To support the sophisticated schedulers of today's frameworks, Mesos introduces a distributed two-level scheduling mechanism called resource offers. Mesos decides *how many* resources to offer each framework, while frameworks decide *which* resources to accept and which computations to run on them. Our results show that Mesos can achieve near-optimal data locality when sharing the cluster among diverse frameworks, can scale to 50,000 (emulated) nodes, and is resilient to failures.

1 Introduction

Clusters of commodity servers have become a major computing platform, powering both large Internet services and a growing number of data-intensive scientific applications. Driven by these applications, researchers and practitioners have been developing a diverse array of cluster computing frameworks to simplify programming the cluster. Prominent examples include MapReduce [18], Dryad [24], MapReduce Online [17] (which supports streaming jobs), Pregel [28] (a specialized framework for graph computations), and others [27, 19, 30].

It seems clear that new cluster computing frameworks[1] will continue to emerge, and that no framework will be optimal for all applications. Therefore, organizations will want to run *multiple frameworks in the same cluster*, picking the best one for each application. Multiplexing a cluster between frameworks improves utilization and allows applications to share access to large datasets that may be too costly to replicate across clusters.

Two common solutions for sharing a cluster today are either to statically partition the cluster and run one framework per partition, or to allocate a set of VMs to each framework. Unfortunately, these solutions achieve neither high utilization nor efficient data sharing. The main problem is the mismatch between the allocation granularities of these solutions and of existing frameworks. Many frameworks, such as Hadoop and Dryad, employ a fine-grained resource sharing model, where nodes are subdivided into "slots" and jobs are composed of short *tasks* that are matched to slots [25, 38]. The short duration of tasks and the ability to run multiple tasks per node allow jobs to achieve high data locality, as each job will quickly get a chance to run on nodes storing its input data. Short tasks also allow frameworks to achieve high utilization, as jobs can rapidly scale when new nodes become available. Unfortunately, because these frameworks are developed independently, there is no way to perform fine-grained sharing *across* frameworks, making it difficult to share clusters and data efficiently between them.

In this paper, we propose Mesos, a thin resource sharing layer that enables fine-grained sharing across diverse cluster computing frameworks, by giving frameworks a common interface for accessing cluster resources.

The main design question for Mesos is how to build a scalable and efficient system that supports a wide array of both current and future frameworks. This is challenging for several reasons. First, each framework will have different scheduling needs, based on its programming model, communication pattern, task dependencies, and data placement. Second, the scheduling system must scale to clusters of tens of thousands of nodes running hundreds of jobs with millions of tasks. Finally, because all the applications in the cluster depend on Mesos, the system must be fault-tolerant and highly available.

One approach would be for Mesos to implement a centralized scheduler that takes as input framework requirements, resource availability, and organizational policies, and computes a global schedule for all tasks. While this

[1] By "framework," we mean a software system that manages and executes one or more jobs on a cluster.

approach can optimize scheduling across frameworks, it faces several challenges. The first is complexity. The scheduler would need to provide a sufficiently expressive API to capture all frameworks' requirements, and to solve an online optimization problem for millions of tasks. Even if such a scheduler were feasible, this complexity would have a negative impact on its scalability and resilience. Second, as new frameworks and new scheduling policies for current frameworks are constantly being developed [37, 38, 40, 26], it is not clear whether we are even at the point to have a full specification of framework requirements. Third, many existing frameworks implement their own sophisticated scheduling [25, 38], and moving this functionality to a global scheduler would require expensive refactoring.

Instead, Mesos takes a different approach: delegating control over scheduling to the frameworks. This is accomplished through a new abstraction, called a *resource offer*, which encapsulates a bundle of resources that a framework can allocate on a cluster node to run tasks. Mesos decides *how many* resources to offer each framework, based on an organizational policy such as fair sharing, while frameworks decide *which* resources to accept and which tasks to run on them. While this decentralized scheduling model may not always lead to globally optimal scheduling, we have found that it performs surprisingly well in practice, allowing frameworks to meet goals such as data locality nearly perfectly. In addition, resource offers are simple and efficient to implement, allowing Mesos to be highly scalable and robust to failures.

Mesos also provides other benefits to practitioners. First, even organizations that only use one framework can use Mesos to run multiple instances of that framework in the same cluster, or multiple versions of the framework. Our contacts at Yahoo! and Facebook indicate that this would be a compelling way to isolate production and experimental Hadoop workloads and to roll out new versions of Hadoop [11, 10]. Second, Mesos makes it easier to develop and immediately experiment with new frameworks. The ability to share resources across multiple frameworks frees the developers to build and run *specialized* frameworks targeted at particular problem domains rather than one-size-fits-all abstractions. Frameworks can therefore evolve faster and provide better support for each problem domain.

We have implemented Mesos in 10,000 lines of C++. The system scales to 50,000 (emulated) nodes and uses ZooKeeper [4] for fault tolerance. To evaluate Mesos, we have ported three cluster computing systems to run over it: Hadoop, MPI, and the Torque batch scheduler. To validate our hypothesis that specialized frameworks provide value over general ones, we have also built a new framework on top of Mesos called Spark, optimized for iterative jobs where a dataset is reused in many parallel operations, and shown that Spark can outperform Hadoop by 10x in iterative machine learning workloads.

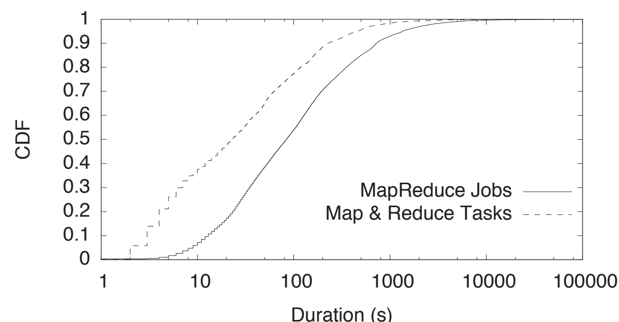


Figure 1: CDF of job and task durations in Facebook's Hadoop data warehouse (data from [38]).

This paper is organized as follows. Section 2 details the data center environment that Mesos is designed for. Section 3 presents the architecture of Mesos. Section 4 analyzes our distributed scheduling model (resource offers) and characterizes the environments that it works well in. We present our implementation of Mesos in Section 5 and evaluate it in Section 6. We survey related work in Section 7. Finally, we conclude in Section 8.

2 Target Environment

As an example of a workload we aim to support, consider the Hadoop data warehouse at Facebook [5]. Facebook loads logs from its web services into a 2000-node Hadoop cluster, where they are used for applications such as business intelligence, spam detection, and ad optimization. In addition to "production" jobs that run periodically, the cluster is used for many experimental jobs, ranging from multi-hour machine learning computations to 1-2 minute ad-hoc queries submitted interactively through an SQL interface called Hive [3]. Most jobs are short (the median job being 84s long), and the jobs are composed of fine-grained map and reduce tasks (the median task being 23s), as shown in Figure 1.

To meet the performance requirements of these jobs, Facebook uses a fair scheduler for Hadoop that takes advantage of the fine-grained nature of the workload to allocate resources at the level of tasks and to optimize data locality [38]. Unfortunately, this means that the cluster can only run Hadoop jobs. If a user wishes to write an ad targeting algorithm in MPI instead of MapReduce, perhaps because MPI is more efficient for this job's communication pattern, then the user must set up a separate MPI cluster and import terabytes of data into it. This problem is not hypothetical; our contacts at Yahoo! and Facebook report that users want to run MPI and MapReduce Online (a streaming MapReduce) [11, 10]. Mesos aims to provide fine-grained sharing between *multiple* cluster computing frameworks to enable these usage scenarios.

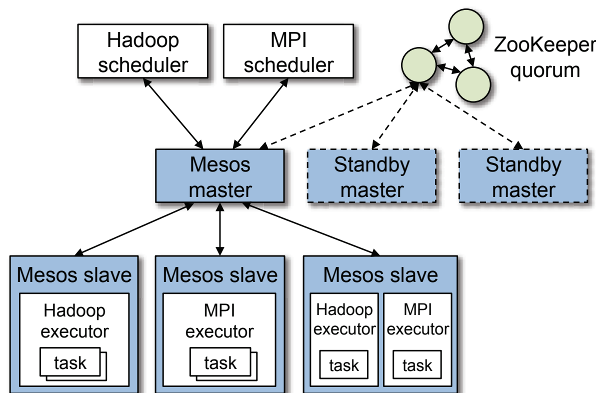


Figure 2: Mesos architecture diagram, showing two running frameworks (Hadoop and MPI).

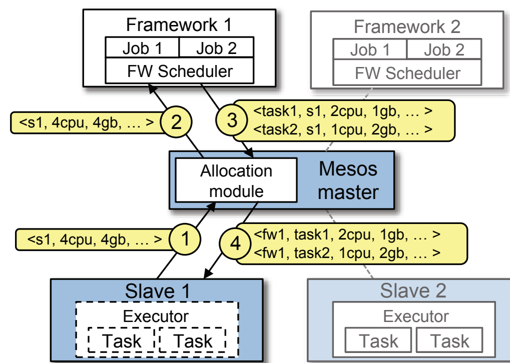


Figure 3: Resource offer example.

3 Architecture

We begin our description of Mesos by discussing our design philosophy. We then describe the components of Mesos, our resource allocation mechanisms, and how Mesos achieves isolation, scalability, and fault tolerance.

3.1 Design Philosophy

Mesos aims to provide a scalable and resilient core for enabling various frameworks to efficiently share clusters. Because cluster frameworks are both highly diverse and rapidly evolving, our overriding design philosophy has been to define a minimal interface that enables efficient resource sharing across frameworks, and otherwise push control of task scheduling and execution to the frameworks. Pushing control to the frameworks has two benefits. First, it allows frameworks to implement diverse approaches to various problems in the cluster (e.g., achieving data locality, dealing with faults), and to evolve these solutions independently. Second, it keeps Mesos simple and minimizes the rate of change required of the system, which makes it easier to keep Mesos scalable and robust.

Although Mesos provides a low-level interface, we expect higher-level libraries implementing common functionality (such as fault tolerance) to be built on top of it. These libraries would be analogous to library OSes in the exokernel [20]. Putting this functionality in libraries rather than in Mesos allows Mesos to remain small and flexible, and lets the libraries evolve independently.

3.2 Overview

Figure 2 shows the main components of Mesos. Mesos consists of a *master* process that manages *slave* daemons running on each cluster node, and *frameworks* that run *tasks* on these slaves.

The master implements fine-grained sharing across frameworks using *resource offers*. Each resource offer is a list of free resources on multiple slaves. The master decides *how many* resources to offer to each framework according to an organizational policy, such as fair sharing or priority. To support a diverse set of inter-framework allocation policies, Mesos lets organizations define their own policies via a pluggable allocation module.

Each framework running on Mesos consists of two components: a *scheduler* that registers with the master to be offered resources, and an *executor* process that is launched on slave nodes to run the framework's tasks. While the master determines how many resources to offer to each framework, the frameworks' schedulers select *which* of the offered resources to use. When a framework accepts offered resources, it passes Mesos a description of the tasks it wants to launch on them.

Figure 3 shows an example of how a framework gets scheduled to run tasks. In step (1), slave 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation module, which tells it that framework 1 should be offered all available resources. In step (2), the master sends a resource offer describing these resources to framework 1. In step (3), the framework's scheduler replies to the master with information about two tasks to run on the slave, using $\langle 2\text{ CPUs}, 1\text{ GB RAM}\rangle$ for the first task, and $\langle 1\text{ CPUs}, 2\text{ GB RAM}\rangle$ for the second task. Finally, in step (4), the master sends the tasks to the slave, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted borders). Because 1 CPU and 1 GB of RAM are still free, the allocation module may now offer them to framework 2. In addition, this resource offer process repeats when tasks finish and new resources become free.

To maintain a thin interface and enable frameworks to evolve independently, Mesos does not *require* frameworks to specify their resource requirements or constraints. Instead, Mesos gives frameworks the ability to *reject* offers. A framework can reject resources that do not satisfy its constraints in order to wait for ones that do. Thus, the rejection mechanism enables frameworks to support arbitrarily complex resource constraints while keeping Mesos simple and scalable.

One potential challenge with solely using the rejec-

tion mechanism to satisfy all framework constraints is efficiency: a framework may have to wait a long time before it receives an offer satisfying its constraints, and Mesos may have to send an offer to many frameworks before one of them accepts it. To avoid this, Mesos also allows frameworks to set *filters*, which are Boolean predicates specifying that a framework will always reject certain resources. For example, a framework might specify a whitelist of nodes it can run on.

There are two points worth noting. First, filters represent just a performance optimization for the resource offer model, as the frameworks still have the ultimate control to reject any resources that they cannot express filters for and to choose which tasks to run on each node. Second, as we will show in this paper, when the workload consists of fine-grained tasks (*e.g.,* in MapReduce and Dryad workloads), the resource offer model performs surprisingly well even in the absence of filters. In particular, we have found that a simple policy called delay scheduling [38], in which frameworks wait for a limited time to acquire nodes storing their data, yields nearly optimal data locality with a wait time of 1-5s.

In the rest of this section, we describe how Mesos performs two key functions: resource allocation (§3.3) and resource isolation (§3.4). We then describe filters and several other mechanisms that make resource offers scalable and robust (§3.5). Finally, we discuss fault tolerance in Mesos (§3.6) and summarize the Mesos API (§3.7).

3.3 Resource Allocation

Mesos delegates allocation decisions to a pluggable allocation module, so that organizations can tailor allocation to their needs. So far, we have implemented two allocation modules: one that performs fair sharing based on a generalization of max-min fairness for multiple resources [21] and one that implements strict priorities. Similar policies are used in Hadoop and Dryad [25, 38].

In normal operation, Mesos takes advantage of the fact that most tasks are short, and only reallocates resources when tasks finish. This usually happens frequently enough so that new frameworks acquire their share quickly. For example, if a framework's share is 10% of the cluster, it needs to wait approximately 10% of the mean task length to receive its share. However, if a cluster becomes filled by long tasks, e.g., due to a buggy job or a greedy framework, the allocation module can also *revoke* (kill) tasks. Before killing a task, Mesos gives its framework a grace period to clean it up.

We leave it up to the allocation module to select the policy for revoking tasks, but describe two related mechanisms here. First, while killing a task has a low impact on many frameworks (*e.g.,* MapReduce), it is harmful for frameworks with interdependent tasks (*e.g.,* MPI). We allow these frameworks to avoid being killed by letting allocation modules expose a *guaranteed allocation* to each framework—a quantity of resources that the framework may hold without losing tasks. Frameworks read their guaranteed allocations through an API call. Allocation modules are responsible for ensuring that the guaranteed allocations they provide can all be met concurrently. For now, we have kept the semantics of guaranteed allocations simple: if a framework is below its guaranteed allocation, none of its tasks should be killed, and if it is above, any of its tasks may be killed.

Second, to decide when to trigger revocation, Mesos must know which of the connected frameworks would use more resources if they were offered them. Frameworks indicate their interest in offers through an API call.

3.4 Isolation

Mesos provides performance isolation between framework executors running on the same slave by leveraging existing OS isolation mechanisms. Since these mechanisms are platform-dependent, we support multiple isolation mechanisms through pluggable *isolation modules*.

We currently isolate resources using OS container technologies, specifically Linux Containers [9] and Solaris Projects [13]. These technologies can limit the CPU, memory, network bandwidth, and (in new Linux kernels) I/O usage of a process tree. These isolation technologies are not perfect, but using containers is already an advantage over frameworks like Hadoop, where tasks from different jobs simply run in separate processes.

3.5 Making Resource Offers Scalable and Robust

Because task scheduling in Mesos is a distributed process, it needs to be efficient and robust to failures. Mesos includes three mechanisms to help with this goal.

First, because some frameworks will always reject certain resources, Mesos lets them short-circuit the rejection process and avoid communication by providing *filters* to the master. We currently support two types of filters: "only offer nodes from list L" and "only offer nodes with at least R resources free". However, other types of predicates could also be supported. Note that unlike generic constraint languages, filters are Boolean predicates that specify whether a framework will reject one bundle of resources on one node, so they can be evaluated quickly on the master. Any resource that does not pass a framework's filter is treated exactly like a rejected resource.

Second, because a framework may take time to respond to an offer, Mesos counts resources offered to a framework towards its allocation of the cluster. This is a strong incentive for frameworks to respond to offers quickly and to filter resources that they cannot use.

Third, if a framework has not responded to an offer for a sufficiently long time, Mesos *rescinds* the offer and re-offers the resources to other frameworks.

Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers) offerRescinded(offerId) statusUpdate(taskId, status) slaveLost(slaveId)	replyToOffer(offerId, tasks) setNeedsOffers(bool) setFilters(filters) getGuaranteedShare() killTask(taskId)
Executor Callbacks	**Executor Actions**
launchTask(taskDescriptor) killTask(taskId)	sendStatus(taskId, status)

Table 1: Mesos API functions for schedulers and executors.

3.6 Fault Tolerance

Since all the frameworks depend on the Mesos master, it is critical to make the master fault-tolerant. To achieve this, we have designed the master to be *soft state*, so that a new master can completely reconstruct its internal state from information held by the slaves and the framework schedulers. In particular, the master's only state is the list of active slaves, active frameworks, and running tasks. This information is sufficient to compute how many resources each framework is using and run the allocation policy. We run multiple masters in a hot-standby configuration using ZooKeeper [4] for leader election. When the active master fails, the slaves and schedulers connect to the next elected master and repopulate its state.

Aside from handling master failures, Mesos reports node failures and executor crashes to frameworks' schedulers. Frameworks can then react to these failures using the policies of their choice.

Finally, to deal with scheduler failures, Mesos allows a framework to register multiple schedulers such that when one fails, another one is notified by the Mesos master to take over. Frameworks must use their own mechanisms to share state between their schedulers.

3.7 API Summary

Table 1 summarizes the Mesos API. The "callback" columns list functions that frameworks must implement, while "actions" are operations that they can invoke.

4 Mesos Behavior

In this section, we study Mesos's behavior for different workloads. Our goal is not to develop an exact model of the system, but to provide a coarse understanding of its behavior, in order to characterize the environments that Mesos's distributed scheduling model works well in.

In short, we find that Mesos performs very well when frameworks can scale up and down elastically, tasks durations are homogeneous, and frameworks prefer all nodes equally (§4.2). When different frameworks prefer different nodes, we show that Mesos can emulate a centralized scheduler that performs fair sharing across frameworks (§4.3). In addition, we show that Mesos can handle heterogeneous task durations without impacting the performance of frameworks with short tasks (§4.4). We also discuss how frameworks are incentivized to improve their performance under Mesos, and argue that these incentives also improve overall cluster utilization (§4.5). We conclude this section with some limitations of Mesos's distributed scheduling model (§4.6).

4.1 Definitions, Metrics and Assumptions

In our discussion, we consider three metrics:

- *Framework ramp-up time:* time it takes a new framework to achieve its allocation (*e.g.,* fair share);
- *Job completion time:* time it takes a job to complete, assuming one job per framework;
- *System utilization:* total cluster utilization.

We characterize workloads along two dimensions: elasticity and task duration distribution. An *elastic* framework, such as Hadoop and Dryad, can scale its resources up and down, *i.e.,* it can start using nodes as soon as it acquires them and release them as soon its task finish. In contrast, a *rigid* framework, such as MPI, can start running its jobs only after it has acquired a fixed quantity of resources, and cannot scale up dynamically to take advantage of new resources or scale down without a large impact on performance. For task durations, we consider both homogeneous and heterogeneous distributions.

We also differentiate between two types of resources: mandatory and preferred. A resource is *mandatory* if a framework must acquire it in order to run. For example, a graphical processing unit (GPU) is mandatory if a framework cannot run without access to GPU. In contrast, a resource is *preferred* if a framework performs "better" using it, but can also run using another equivalent resource. For example, a framework may prefer running on a node that locally stores its data, but may also be able to read the data remotely if it must.

We assume the amount of mandatory resources requested by a framework never exceeds its guaranteed share. This ensures that frameworks will not deadlock waiting for the mandatory resources to become free.[2] For simplicity, we also assume that all tasks have the same resource demands and run on identical slices of machines called *slots*, and that each framework runs a single job.

4.2 Homogeneous Tasks

We consider a cluster with n slots and a framework, f, that is entitled to k slots. For the purpose of this analysis, we consider two distributions of the task durations: constant (*i.e.,* all tasks have the same length) and exponential. Let the mean task duration be T, and assume that

[2] In workloads where the mandatory resource demands of the active frameworks can exceed the capacity of the cluster, the allocation module needs to implement admission control.

	Elastic Framework		Rigid Framework	
	Constant dist.	Exponential dist.	Constant dist.	Exponential dist.
Ramp-up time	T	$T \ln k$	T	$T \ln k$
Completion time	$(1/2+\beta)T$	$(1+\beta)T$	$(1+\beta)T$	$(\ln k+\beta)T$
Utilization	1	1	$\beta/(1/2+\beta)$	$\beta/(\ln k-1+\beta)$

Table 2: Ramp-up time, job completion time and utilization for both elastic and rigid frameworks, and for both constant and exponential task duration distributions. The framework starts with no slots. k is the number of slots the framework is entitled under the scheduling policy, and βT represents the time it takes a job to complete assuming the framework gets all k slots at once.

framework f runs a job which requires βkT total computation time. That is, when the framework has k slots, it takes its job βT time to finish.

Table 2 summarizes the job completion times and system utilization for the two types of frameworks and the two types of task length distributions. As expected, elastic frameworks with constant task durations perform the best, while rigid frameworks with exponential task duration perform the worst. Due to lack of space, we present only the results here and include derivations in [23].

Framework ramp-up time: If task durations are constant, it will take framework f at most T time to acquire k slots. This is simply because during a T interval, every slot will become available, which will enable Mesos to offer the framework all k of its preferred slots. If the duration distribution is exponential, the expected ramp-up time can be as high as $T \ln k$ [23].

Job completion time: The expected completion time[3] of an elastic job is at most $(1+\beta)T$, which is within T (*i.e.,* the mean task duration) of the completion time of the job when it gets all its slots instantaneously. Rigid jobs achieve similar completion times for constant task durations, but exhibit much higher completion times for exponential job durations, *i.e.,* $(\ln k+\beta)T$. This is simply because it takes a framework $T \ln k$ time on average to acquire all its slots and be able to start its job.

System utilization: Elastic jobs fully utilize their allocated slots, because they can use every slot as soon as they get it. As a result, assuming infinite demand, a system running only elastic jobs is fully utilized. Rigid frameworks achieve slightly worse utilizations, as their jobs cannot start before they get their full allocations, and thus they waste the resources held while ramping up.

4.3 Placement Preferences

So far, we have assumed that frameworks have no slot preferences. In practice, different frameworks prefer different nodes and their preferences may change over time. In this section, we consider the case where frameworks have different preferred slots.

The natural question is how well Mesos will work compared to a central scheduler that has full information about framework preferences. We consider two cases: (a) there exists a system configuration in which each framework gets all its preferred slots and achieves its full allocation, and (b) there is no such configuration, *i.e.,* the demand for some preferred slots exceeds the supply.

In the first case, it is easy to see that, irrespective of the initial configuration, the system will converge to the state where each framework allocates its preferred slots after at most one T interval. This is simple because during a T interval all slots become available, and as a result each framework will be offered its preferred slots.

In the second case, there is no configuration in which all frameworks can satisfy their preferences. The key question in this case is how should one allocate the preferred slots across the frameworks demanding them. In particular, assume there are p slots preferred by m frameworks, where framework i requests r_i such slots, and $\sum_{i=1}^{m} r_i > x$. While many allocation policies are possible, here we consider a weighted fair allocation policy where the weight associated with framework i is its intended total allocation, s_i. In other words, assuming that each framework has enough demand, we aim to allocate $p{\cdot}s_i/(\sum_{i=1}^{m} s_i)$ preferred slots to framework i.

The challenge in Mesos is that the scheduler does not know the preferences of each framework. Fortunately, it turns out that there is an easy way to achieve the weighted allocation of the preferred slots described above: simply perform lottery scheduling [36], offering slots to frameworks with probabilities proportional to their intended allocations. In particular, when a slot becomes available, Mesos can offer that slot to framework i with probability $s_i/(\sum_{i=1}^{n} s_i)$, where n is the total number of frameworks in the system. Furthermore, because each framework i receives on average s_i slots every T time units, the results for ramp-up times and completion times in Section 4.2 still hold.

4.4 Heterogeneous Tasks

So far we have assumed that frameworks have homogeneous task duration distributions, *i.e.,* that all frameworks have the same task duration distribution. In this section, we discuss frameworks with heterogeneous task duration distributions. In particular, we consider a workload where tasks that are either short and long, where the mean duration of the long tasks is significantly longer

[3] When computing job completion time we assume that the last tasks of the job running on the framework's k slots finish at the same time.

than the mean of the short tasks. Such heterogeneous workloads can hurt frameworks with short tasks. In the worst case, all nodes required by a short job might be filled with long tasks, so the job may need to wait a long time (relative to its execution time) to acquire resources.

We note first that random task assignment can work well if the fraction ϕ of long tasks is not very close to 1 and if each node supports multiple slots. For example, in a cluster with S slots per node, the probability that a node is filled with long tasks will be ϕ^S. When S is large (*e.g.,* in the case of multicore machines), this probability is small even with $\phi > 0.5$. If $S = 8$ and $\phi = 0.5$, for example, the probability that a node is filled with long tasks is 0.4%. Thus, a framework with short tasks can still acquire many preferred slots in a short period of time. In addition, the more slots a framework is able to use, the likelier it is that at least k of them are running short tasks.

To further alleviate the impact of long tasks, Mesos can be extended slightly to allow allocation policies to reserve some resources on each node for short tasks. In particular, we can associate a maximum task duration with some of the resources on each node, after which tasks running on those resources are killed. These time limits can be exposed to the frameworks in resource offers, allowing them to choose whether to use these resources. This scheme is similar to the common policy of having a separate queue for short jobs in HPC clusters.

4.5 Framework Incentives

Mesos implements a decentralized scheduling model, where each framework decides which offers to accept. As with any decentralized system, it is important to understand the incentives of entities in the system. In this section, we discuss the incentives of frameworks (and their users) to improve the response times of their jobs.

Short tasks: A framework is incentivized to use short tasks for two reasons. First, it will be able to allocate any resources reserved for short slots. Second, using small tasks minimizes the wasted work if the framework loses a task, either due to revocation or simply due to failures.

Scale elastically: The ability of a framework to use resources as soon as it acquires them–instead of waiting to reach a given minimum allocation–would allow the framework to start (and complete) its jobs earlier. In addition, the ability to scale up and down allows a framework to grab unused resources opportunistically, as it can later release them with little negative impact.

Do not accept unknown resources: Frameworks are incentivized not to accept resources that they cannot use because most allocation policies will count all the resources that a framework owns when making offers.

We note that these incentives align well with our goal of improving utilization. If frameworks use short tasks, Mesos can reallocate resources quickly between them, reducing latency for new jobs and wasted work for revocation. If frameworks are elastic, they will opportunistically utilize all the resources they can obtain. Finally, if frameworks do not accept resources that they do not understand, they will leave them for frameworks that do.

We also note that these properties are met by many current cluster computing frameworks, such as MapReduce and Dryad, simply because using short independent tasks simplifies load balancing and fault recovery.

4.6 Limitations of Distributed Scheduling

Although we have shown that distributed scheduling works well in a range of workloads relevant to current cluster environments, like any decentralized approach, it can perform worse than a centralized scheduler. We have identified three limitations of the distributed model:

Fragmentation: When tasks have heterogeneous resource demands, a distributed collection of frameworks may not be able to optimize bin packing as well as a centralized scheduler. However, note that the wasted space due to suboptimal bin packing is bounded by the ratio between the largest task size and the node size. Therefore, clusters running "larger" nodes (e.g., multicore nodes) and "smaller" tasks within those nodes will achieve high utilization even with distributed scheduling.

There is another possible bad outcome if allocation modules reallocate resources in a naïve manner: when a cluster is filled by tasks with small resource requirements, a framework f with large resource requirements may starve, because whenever a small task finishes, f cannot accept the resources freed by it, but other frameworks can. To accommodate frameworks with large per-task resource requirements, allocation modules can support a *minimum offer size* on each slave, and abstain from offering resources on the slave until this amount is free.

Interdependent framework constraints: It is possible to construct scenarios where, because of esoteric interdependencies between frameworks (*e.g.,* certain tasks from two frameworks cannot be colocated), only a single global allocation of the cluster performs well. We argue such scenarios are rare in practice. In the model discussed in this section, where frameworks only have preferences over which nodes they use, we showed that allocations approximate those of optimal schedulers.

Framework complexity: Using resource offers may make framework scheduling more complex. We argue, however, that this difficulty is not onerous. First, whether using Mesos or a centralized scheduler, frameworks need to know their preferences; in a centralized scheduler, the framework needs to express them to the scheduler, whereas in Mesos, it must use them to decide which offers to accept. Second, many scheduling policies for ex-

isting frameworks are online algorithms, because frameworks cannot predict task times and must be able to handle failures and stragglers [18, 40, 38]. These policies are easy to implement over resource offers.

5 Implementation

We have implemented Mesos in about 10,000 lines of C++. The system runs on Linux, Solaris and OS X, and supports frameworks written in C++, Java, and Python.

To reduce the complexity of our implementation, we use a C++ library called `libprocess` [7] that provides an actor-based programming model using efficient asynchronous I/O mechanisms (`epoll`, `kqueue`, etc). We also use ZooKeeper [4] to perform leader election.

Mesos can use Linux containers [9] or Solaris projects [13] to isolate tasks. We currently isolate CPU cores and memory. We plan to leverage recently added support for network and I/O isolation in Linux [8] in the future.

We have implemented four frameworks on top of Mesos. First, we have ported three existing cluster computing systems: Hadoop [2], the Torque resource scheduler [33], and the MPICH2 implementation of MPI [16]. None of these ports required changing these frameworks' APIs, so all of them can run unmodified user programs. In addition, we built a specialized framework for iterative jobs called Spark, which we discuss in Section 5.3.

5.1 Hadoop Port

Porting Hadoop to run on Mesos required relatively few modifications, because Hadoop's fine-grained map and reduce tasks map cleanly to Mesos tasks. In addition, the Hadoop master, known as the JobTracker, and Hadoop slaves, known as TaskTrackers, fit naturally into the Mesos model as a framework scheduler and executor.

To add support for running Hadoop on Mesos, we took advantage of the fact that Hadoop already has a pluggable API for writing job schedulers. We wrote a Hadoop scheduler that connects to Mesos, launches TaskTrackers as its executors, and maps each Hadoop task to a Mesos task. When there are unlaunched tasks in Hadoop, our scheduler first starts Mesos tasks on the nodes of the cluster that it wants to use, and then sends the Hadoop tasks to them using Hadoop's existing internal interfaces. When tasks finish, our executor notifies Mesos by listening for task finish events using an API in the TaskTracker.

We used delay scheduling [38] to achieve data locality by waiting for slots on the nodes that contain task input data. In addition, our approach allowed us to reuse Hadoop's existing logic for re-scheduling of failed tasks and for speculative execution (straggler mitigation).

We also needed to change how map output data is served to reduce tasks. Hadoop normally writes map output files to the local filesystem, then serves these to reduce tasks using an HTTP server included in the TaskTracker. However, the TaskTracker within Mesos runs as an executor, which may be terminated if it is not running tasks. This would make map output files unavailable to reduce tasks. We solved this problem by providing a shared file server on each node in the cluster to serve local files. Such a service is useful beyond Hadoop, to other frameworks that write data locally on each node.

In total, our Hadoop port is 1500 lines of code.

5.2 Torque and MPI Ports

We have ported the Torque cluster resource manager to run as a framework on Mesos. The framework consists of a Mesos scheduler and executor, written in 360 lines of Python code, that launch and manage different components of Torque. In addition, we modified 3 lines of Torque source code to allow it to elastically scale up and down on Mesos depending on the jobs in its queue.

After registering with the Mesos master, the framework scheduler configures and launches a Torque server and then periodically monitors the server's job queue. While the queue is empty, the scheduler releases all tasks (down to an optional minimum, which we set to 0) and refuses all resource offers it receives from Mesos. Once a job gets added to Torque's queue (using the standard `qsub` command), the scheduler begins accepting new resource offers. As long as there are jobs in Torque's queue, the scheduler accepts offers as necessary to satisfy the constraints of as many jobs in the queue as possible. On each node where offers are accepted, Mesos launches our executor, which in turn starts a Torque backend daemon and registers it with the Torque server. When enough Torque backend daemons have registered, the torque server will launch the next job in its queue.

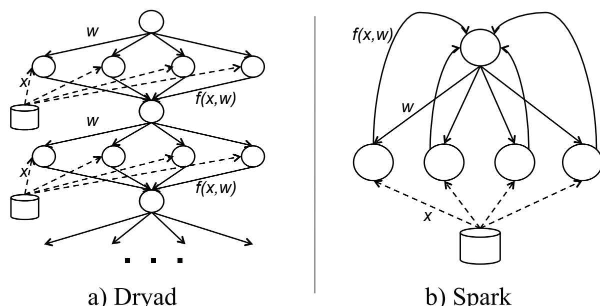
Because jobs that run on Torque (e.g. MPI) may not be fault tolerant, Torque avoids having its tasks revoked by not accepting resources beyond its guaranteed allocation.

In addition to the Torque framework, we also created a Mesos MPI "wrapper" framework, written in 200 lines of Python code, for running MPI jobs directly on Mesos.

5.3 Spark Framework

Mesos enables the creation of specialized frameworks optimized for workloads for which more general execution layers may not be optimal. To test the hypothesis that simple specialized frameworks provide value, we identified one class of jobs that were found to perform poorly on Hadoop by machine learning researchers at our lab: *iterative jobs*, where a dataset is reused across a number of iterations. We built a specialized framework called Spark [39] optimized for these workloads.

One example of an iterative algorithm used in machine learning is logistic regression [22]. This algorithm seeks to find a line that separates two sets of labeled data points. The algorithm starts with a random line w. Then,



a) Dryad b) Spark

Figure 4: Data flow of a logistic regression job in Dryad vs. Spark. Solid lines show data flow within the framework. Dashed lines show reads from a distributed file system. Spark reuses in-memory data across iterations to improve efficiency.

on each iteration, it computes the gradient of an objective function that measures how well the line separates the points, and shifts w along this gradient. This gradient computation amounts to evaluating a function $f(x, w)$ over each data point x and summing the results. An implementation of logistic regression in Hadoop must run each iteration as a separate MapReduce job, because each iteration depends on the w computed at the previous one. This imposes overhead because every iteration must re-read the input file into memory. In Dryad, the whole job can be expressed as a data flow DAG as shown in Figure 4a, but the data must still must be reloaded from disk at each iteration. Reusing the data in memory between iterations in Dryad would require cyclic data flow.

Spark's execution is shown in Figure 4b. Spark uses the long-lived nature of Mesos executors to cache a slice of the dataset in memory at each executor, and then run multiple iterations on this cached data. This caching is achieved in a fault-tolerant manner: if a node is lost, Spark remembers how to recompute its slice of the data.

By building Spark on top of Mesos, we were able to keep its implementation small (about 1300 lines of code), yet still capable of outperforming Hadoop by $10\times$ for iterative jobs. In particular, using Mesos's API saved us the time to write a master daemon, slave daemon, and communication protocols between them for Spark. The main pieces we had to write were a framework scheduler (which uses delay scheduling for locality) and user APIs.

6 Evaluation

We evaluated Mesos through a series of experiments on the Amazon Elastic Compute Cloud (EC2). We begin with a macrobenchmark that evaluates how the system shares resources between four workloads, and go on to present a series of smaller experiments designed to evaluate overhead, decentralized scheduling, our specialized framework (Spark), scalability, and failure recovery.

Bin	Job Type	Map Tasks	Reduce Tasks	# Jobs Run
1	selection	1	NA	38
2	text search	2	NA	18
3	aggregation	10	2	14
4	selection	50	NA	12
5	aggregation	100	10	6
6	selection	200	NA	6
7	text search	400	NA	4
8	join	400	30	2

Table 3: Job types for each bin in our Facebook Hadoop mix.

6.1 Macrobenchmark

To evaluate the primary goal of Mesos, which is enabling diverse frameworks to efficiently share a cluster, we ran a macrobenchmark consisting of a mix of four workloads:

- A Hadoop instance running a mix of small and large jobs based on the workload at Facebook.
- A Hadoop instance running a set of large batch jobs.
- Spark running a series of machine learning jobs.
- Torque running a series of MPI jobs.

We compared a scenario where the workloads ran as four frameworks on a 96-node Mesos cluster using fair sharing to a scenario where they were each given a static partition of the cluster (24 nodes), and measured job response times and resource utilization in both cases. We used EC2 nodes with 4 CPU cores and 15 GB of RAM.

We begin by describing the four workloads in more detail, and then present our results.

6.1.1 Macrobenchmark Workloads

Facebook Hadoop Mix Our Hadoop job mix was based on the distribution of job sizes and inter-arrival times at Facebook, reported in [38]. The workload consists of 100 jobs submitted at fixed times over a 25-minute period, with a mean inter-arrival time of 14s. Most of the jobs are small (1-12 tasks), but there are also large jobs of up to 400 tasks.[4] The jobs themselves were from the Hive benchmark [6], which contains four types of queries: text search, a simple selection, an aggregation, and a join that gets translated into multiple MapReduce steps. We grouped the jobs into eight bins of job type and size (listed in Table 3) so that we could compare performance in each bin. We also set the framework scheduler to perform fair sharing between its jobs, as this policy is used at Facebook.

Large Hadoop Mix To emulate batch workloads that need to run continuously, such as web crawling, we had a second instance of Hadoop run a series of IO-intensive 2400-task text search jobs. A script launched ten of these jobs, submitting each one after the previous one finished.

[4] We scaled down the largest jobs in [38] to have the workload fit a quarter of our cluster size.

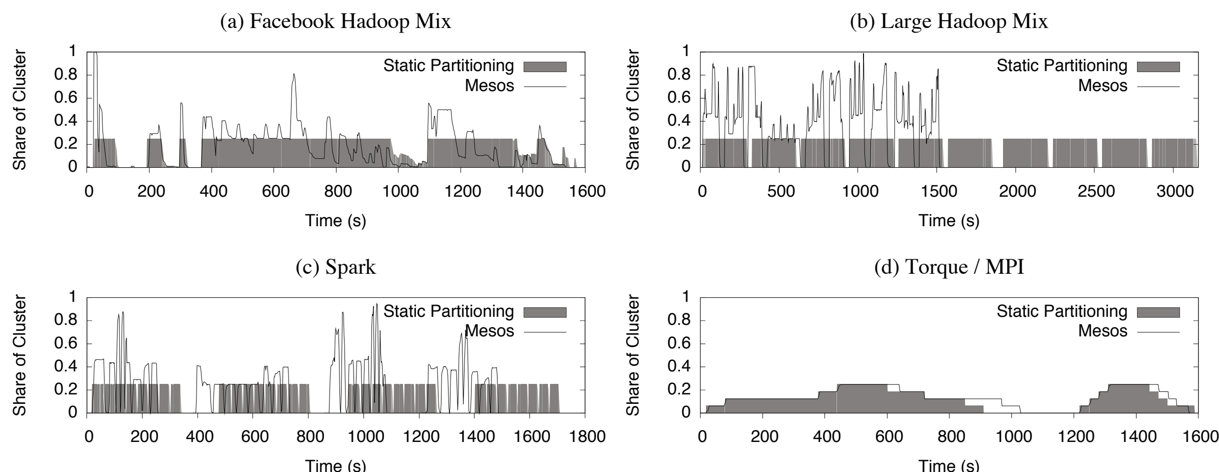


Figure 5: Comparison of cluster shares (fraction of CPUs) over time for each of the frameworks in the Mesos and static partitioning macrobenchmark scenarios. On Mesos, frameworks can scale up when their demand is high and that of other frameworks is low, and thus finish jobs faster. Note that the plots' time axes are different (*e.g.,* the large Hadoop mix takes 3200s with static partitioning).

Spark We ran five instances of an iterative machine learning job on Spark. These were launched by a script that waited 2 minutes after each job ended to submit the next. The job we used was alternating least squares (ALS), a collaborative filtering algorithm [42]. This job is CPU-intensive but also benefits from caching its input data on each node, and needs to broadcast updated parameters to all nodes running its tasks on each iteration.

Torque / MPI Our Torque framework ran eight instances of the `tachyon` raytracing job [35] that is part of the SPEC MPI2007 benchmark. Six of the jobs ran small problem sizes and two ran large ones. Both types used 24 parallel tasks. We submitted these jobs at fixed times to both clusters. The `tachyon` job is CPU-intensive.

6.1.2 Macrobenchmark Results

A successful result for Mesos would show two things: that Mesos achieves higher utilization than static partitioning, and that jobs finish at least as fast in the shared cluster as they do in their static partition, and possibly faster due to gaps in the demand of other frameworks. Our results show both effects, as detailed below.

We show the fraction of CPU cores allocated to each framework by Mesos over time in Figure 6. We see that Mesos enables each framework to scale up during periods when other frameworks have low demands, and thus keeps cluster nodes busier. For example, at time 350, when both Spark and the Facebook Hadoop framework have no running jobs and Torque is using 1/8 of the cluster, the large-job Hadoop framework scales up to 7/8 of the cluster. In addition, we see that resources are reallocated rapidly (*e.g.,* when a Facebook Hadoop job starts around time 360) due to the fine-grained nature of tasks. Finally, higher allocation of nodes also translates into increased CPU and memory utilization (by 10% for CPU and 17% for memory), as shown in Figure 7.

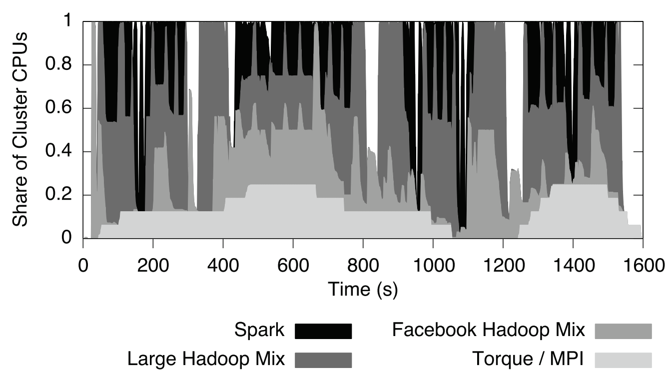


Figure 6: Framework shares on Mesos during the macrobenchmark. By pooling resources, Mesos lets each workload scale up to fill gaps in the demand of others. In addition, fine-grained sharing allows resources to be reallocated in tens of seconds.

A second question is how much better jobs perform under Mesos than when using a statically partitioned cluster. We present this data in two ways. First, Figure 5 compares the resource allocation over time of each framework in the shared and statically partitioned clusters. Shaded areas show the allocation in the statically partitioned cluster, while solid lines show the share on Mesos. We see that the fine-grained frameworks (Hadoop and Spark) take advantage of Mesos to scale up beyond 1/4 of the cluster when global demand allows this, and consequently finish bursts of submitted jobs faster in Mesos. At the same time, Torque achieves roughly similar allocations and job durations under Mesos (with some differences explained later).

Second, Tables 4 and 5 show a breakdown of job performance for each framework. In Table 4, we compare the aggregate performance of each framework, defined as the sum of job running times, in the static partitioning

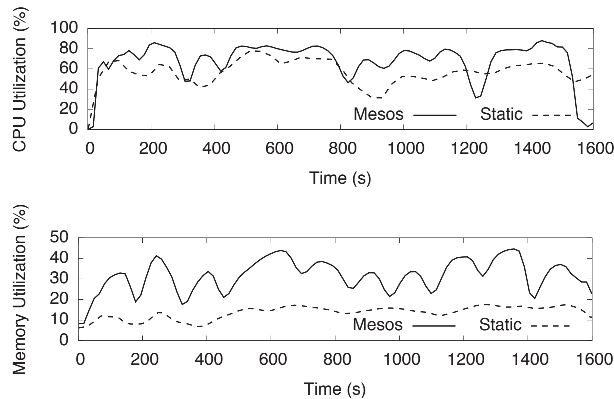


Figure 7: Average CPU and memory utilization over time across all nodes in the Mesos cluster vs. static partitioning.

Framework	Sum of Exec Times w/ Static Partitioning (s)	Sum of Exec Times with Mesos (s)	Speedup
Facebook Hadoop Mix	7235	6319	**1.14**
Large Hadoop Mix	3143	1494	**2.10**
Spark	1684	1338	**1.26**
Torque / MPI	3210	3352	**0.96**

Table 4: Aggregate performance of each framework in the macrobenchmark (sum of running times of all the jobs in the framework). The speedup column shows the relative gain on Mesos.

and Mesos scenarios. We see the Hadoop and Spark jobs as a whole are finishing faster on Mesos, while Torque is slightly slower. The framework that gains the most is the large-job Hadoop mix, which almost always has tasks to run and fills in the gaps in demand of the other frameworks; this framework performs 2x better on Mesos.

Table 5 breaks down the results further by job type. We observe two notable trends. First, in the Facebook Hadoop mix, the smaller jobs perform worse on Mesos. This is due to an interaction between the fair sharing performed by Hadoop (among its jobs) and the fair sharing in Mesos (among frameworks): During periods of time when Hadoop has more than 1/4 of the cluster, if any jobs are submitted to the other frameworks, there is a delay before Hadoop gets a new resource offer (because any freed up resources go to the framework farthest below its share), so any small job submitted during this time is delayed for a long time relative to its length. In contrast, when running alone, Hadoop can assign resources to the new job as soon as any of its tasks finishes. This problem with hierarchical fair sharing is also seen in networks [34], and could be mitigated by running the small jobs on a separate framework or using a different allocation policy (*e.g.,* using lottery scheduling instead of offering all freed resources to the framework with the lowest share).

Lastly, Torque is the only framework that performed worse, on average, on Mesos. The large `tachyon` jobs took on average 2 minutes longer, while the small ones

Framework	Job Type	Exec Time w/ Static Partitioning (s)	Avg. Speedup on Mesos
Facebook Hadoop Mix	selection (1)	24	**0.84**
	text search (2)	31	**0.90**
	aggregation (3)	82	**0.94**
	selection (4)	65	**1.40**
	aggregation (5)	192	**1.26**
	selection (6)	136	**1.71**
	text search (7)	137	**2.14**
	join (8)	662	**1.35**
Large Hadoop Mix	text search	314	**2.21**
Spark	ALS	337	**1.36**
Torque / MPI	small tachyon	261	**0.91**
	large tachyon	822	**0.88**

Table 5: Performance of each job type in the macrobenchmark. Bins for the Facebook Hadoop mix are in parentheses.

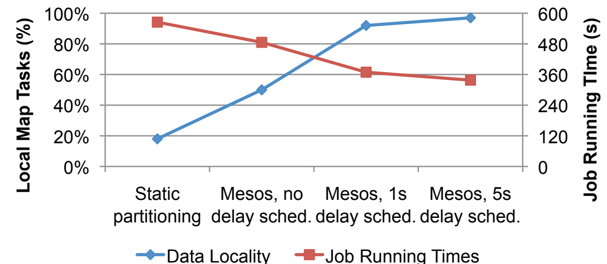


Figure 8: Data locality and average job durations for 16 Hadoop instances running on a 93-node cluster using static partitioning, Mesos, or Mesos with delay scheduling.

took 20s longer. Some of this delay is due to Torque having to wait to launch 24 tasks on Mesos before starting each job, but the average time this takes is 12s. We believe that the rest of the delay is due to stragglers (slow nodes). In our standalone Torque run, we saw two jobs take about 60s longer to run than the others (Fig. 5d). We discovered that both of these jobs were using a node that performed slower on single-node benchmarks than the others (in fact, Linux reported 40% lower bogomips on it). Because `tachyon` hands out equal amounts of work to each node, it runs as slowly as the slowest node.

6.2 Overhead

To measure the overhead Mesos imposes when a single framework uses the cluster, we ran two benchmarks using MPI and Hadoop on an EC2 cluster with 50 nodes, each with 2 CPU cores and 6.5 GB RAM. We used the High-Performance LINPACK [15] benchmark for MPI and a WordCount job for Hadoop, and ran each job three times. The MPI job took on average 50.9s without Mesos and 51.8s with Mesos, while the Hadoop job took 160s without Mesos and 166s with Mesos. In both cases, the overhead of using Mesos was less than 4%.

6.3 Data Locality through Delay Scheduling

In this experiment, we evaluated how Mesos' resource offer mechanism enables frameworks to control their

tasks' placement, and in particular, data locality. We ran 16 instances of Hadoop using 93 EC2 nodes, each with 4 CPU cores and 15 GB RAM. Each node ran a map-only scan job that searched a 100 GB file spread throughout the cluster on a shared HDFS file system and outputted 1% of the records. We tested four scenarios: giving each Hadoop instance its own 5-6 node static partition of the cluster (to emulate organizations that use coarse-grained cluster sharing systems), and running all instances on Mesos using either no delay scheduling, 1s delay scheduling or 5s delay scheduling.

Figure 8 shows averaged measurements from the 16 Hadoop instances across three runs of each scenario. Using static partitioning yields very low data locality (18%) because the Hadoop instances are forced to fetch data from nodes outside their partition. In contrast, running the Hadoop instances on Mesos improves data locality, even without delay scheduling, because each Hadoop instance has tasks on more nodes of the cluster (there are 4 tasks per node), and can therefore access more blocks locally. Adding a 1-second delay brings locality above 90%, and a 5-second delay achieves 95% locality, which is competitive with running one Hadoop instance alone on the whole cluster. As expected, job performance improves with data locality: jobs run 1.7x faster in the 5s delay scenario than with static partitioning.

6.4 Spark Framework

We evaluated the benefit of running iterative jobs using the specialized Spark framework we developed on top of Mesos (Section 5.3) over the general-purpose Hadoop framework. We used a logistic regression job implemented in Hadoop by machine learning researchers in our lab, and wrote a second version of the job using Spark. We ran each version separately on 20 EC2 nodes, each with 4 CPU cores and 15 GB RAM. Each experiment used a 29 GB data file and varied the number of logistic regression iterations from 1 to 30 (see Figure 9).

With Hadoop, each iteration takes 127s on average, because it runs as a separate MapReduce job. In contrast, with Spark, the first iteration takes 174s, but subsequent iterations only take about 6 seconds, leading to a speedup of up to 10x for 30 iterations. This happens because the cost of reading the data from disk and parsing it is much higher than the cost of evaluating the gradient function computed by the job on each iteration. Hadoop incurs the read/parsing cost on each iteration, while Spark reuses cached blocks of parsed data and only incurs this cost once. The longer time for the first iteration in Spark is due to the use of slower text parsing routines.

6.5 Mesos Scalability

To evaluate Mesos' scalability, we emulated large clusters by running up to 50,000 slave daemons on 99 Amazon EC2 nodes, each with 8 CPU cores and 6 GB RAM. We used one EC2 node for the master and the rest of the nodes to run slaves. During the experiment, each of 200 frameworks running throughout the cluster continuously launches tasks, starting one task on each slave that it receives a resource offer for. Each task sleeps for a period of time based on a normal distribution with a mean of 30 seconds and standard deviation of 10s, and then ends. Each slave runs up to two tasks at a time.

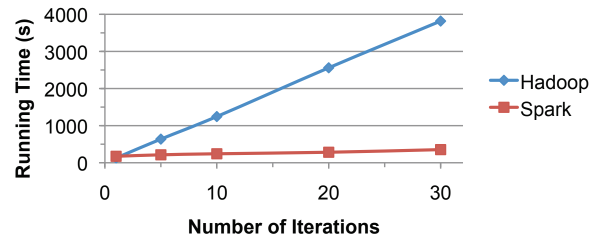


Figure 9: Hadoop and Spark logistic regression running times.

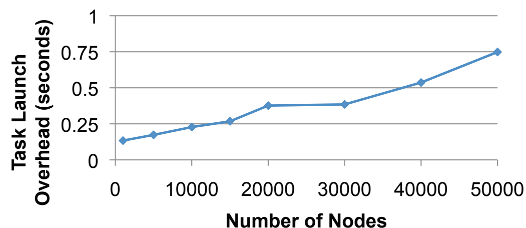


Figure 10: Mesos master's scalability versus number of slaves.

Once the cluster reached steady-state (i.e., the 200 frameworks achieve their fair shares and all resources were allocated), we launched a test framework that runs a single 10 second task and measured how long this framework took to finish. This allowed us to calculate the extra delay incurred over 10s due to having to register with the master, wait for a resource offer, accept it, wait for the master to process the response and launch the task on a slave, and wait for Mesos to report the task as finished.

We plot this extra delay in Figure 10, showing averages of 5 runs. We observe that the overhead remains small (less than one second) even at 50,000 nodes. In particular, this overhead is much smaller than the average task and job lengths in data center workloads (see Section 2). Because Mesos was also keeping the cluster fully allocated, this indicates that the master kept up with the load placed on it. Unfortunately, the EC2 virtualized environment limited scalability beyond 50,000 slaves, because at 50,000 slaves the master was processing 100,000 packets per second (in+out), which has been shown to be the current achievable limit on EC2 [12].

6.6 Failure Recovery

To evaluate recovery from master failures, we conducted an experiment with 200 to 4000 slave daemons on 62

EC2 nodes with 4 cores and 15 GB RAM. We ran 200 frameworks that each launched 20-second tasks, and two Mesos masters connected to a 5-node ZooKeeper quorum.We synchronized the two masters' clocks using NTP and measured the mean time to recovery (MTTR) after killing the active master. The MTTR is the time for all of the slaves and frameworks to connect to the second master. In all cases, the MTTR was between 4 and 8 seconds, with 95% confidence intervals of up to 3s on either side.

6.7 Performance Isolation

As discussed in Section 3.4, Mesos leverages existing OS isolation mechanism to provide performance isolation between different frameworks' tasks running on the same slave. While these mechanisms are not perfect, a preliminary evaluation of Linux Containers [9] shows promising results. In particular, using Containers to isolate CPU usage between a MediaWiki web server (consisting of multiple Apache processes running PHP) and a "hog" application (consisting of 256 processes spinning in infinite loops) shows on average only a 30% increase in request latency for Apache versus a 550% increase when running without Containers. We refer the reader to [29] for a fuller evaluation of OS isolation mechanisms.

7 Related Work

HPC and Grid Schedulers. The high performance computing (HPC) community has long been managing clusters [33, 41]. However, their target environment typically consists of specialized hardware, such as Infiniband and SANs, where jobs do not need to be scheduled local to their data. Furthermore, each job is tightly coupled, often using barriers or message passing. Thus, each job is monolithic, rather than composed of fine-grained tasks, and does not change its resource demands during its lifetime. For these reasons, HPC schedulers use centralized scheduling, and require users to declare the required resources at job submission time. Jobs are then given coarse-grained allocations of the cluster. Unlike the Mesos approach, this does not allow jobs to locally access data distributed across the cluster. Furthermore, jobs cannot grow and shrink dynamically. In contrast, Mesos supports fine-grained sharing at the level of tasks and allows frameworks to control their placement.

Grid computing has mostly focused on the problem of making diverse virtual organizations share geographically distributed and separately administered resources in a secure and interoperable way. Mesos could well be used within a virtual organization inside a larger grid.

Public and Private Clouds. Virtual machine clouds such as Amazon EC2 [1] and Eucalyptus [31] share common goals with Mesos, such as isolating applications while providing a low-level abstraction (VMs). However, they differ from Mesos in several important ways. First, their relatively coarse grained VM allocation model leads to less efficient resource utilization and data sharing than in Mesos. Second, these systems generally do not let applications specify placement needs beyond the size of VM they require. In contrast, Mesos allows frameworks to be highly selective about task placement.

Quincy. Quincy [25] is a fair scheduler for Dryad that uses a centralized scheduling algorithm for Dryad's DAG-based programming model. In contrast, Mesos provides the lower-level abstraction of resource offers to support *multiple* cluster computing frameworks.

Condor. The Condor cluster manager uses the ClassAds language [32] to match nodes to jobs. Using a resource specification language is not as flexible for frameworks as resource offers, since not all requirements may be expressible. Also, porting existing frameworks, which have their own schedulers, to Condor would be more difficult than porting them to Mesos, where existing schedulers fit naturally into the two-level scheduling model.

Next-Generation Hadoop. In February 2011, Yahoo! announced a redesign for Hadoop that uses a two-level scheduling model, where per-application masters request resources from a central manager [14]. The design aims to support non-MapReduce applications too. While details about the scheduling model in this system are currently unavailable, we believe that the new application masters could naturally run as Mesos frameworks.

8 Conclusion and Future Work

We have presented Mesos, a thin management layer that allows diverse cluster computing frameworks to efficiently share resources. Mesos is built around two design elements: a fine-grained sharing model at the level of tasks, and a distributed scheduling mechanism called resource offers that delegates scheduling decisions to the frameworks. Together, these elements let Mesos achieve high utilization, respond quickly to workload changes, and cater to diverse frameworks while remaining scalable and robust. We have shown that existing frameworks can effectively share resources using Mesos, that Mesos enables the development of specialized frameworks providing major performance gains, such as Spark, and that Mesos's simple design allows the system to be fault tolerant and to scale to 50,000 nodes.

In future work, we plan to further analyze the resource offer model and determine whether any extensions can improve its efficiency while retaining its flexibility. In particular, it may be possible to have frameworks give richer hints about offers they would like to receive. Nonetheless, we believe that below any hint system, frameworks should still have the ability to reject offers and to choose which tasks to launch on each resource, so that their evolution is not constrained by the

hint language provided by the system.

We are also currently using Mesos to manage resources on a 40-node cluster in our lab and in a test deployment at Twitter, and plan to report on lessons from these deployments in future work.

9 Acknowledgements

We thank our industry colleagues at Google, Twitter, Facebook, Yahoo! and Cloudera for their valuable feedback on Mesos. This research was supported by California MICRO, California Discovery, the Natural Sciences and Engineering Research Council of Canada, a National Science Foundation Graduate Research Fellowship,[5] the Swedish Research Council, and the following Berkeley RAD Lab sponsors: Google, Microsoft, Oracle, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.
[2] Apache Hadoop. http://hadoop.apache.org.
[3] Apache Hive. http://hadoop.apache.org/hive.
[4] Apache ZooKeeper. hadoop.apache.org/zookeeper.
[5] Hive – A Petabyte Scale Data Warehouse using Hadoop. http://www.facebook.com/note.php?note_id=89508453919.
[6] Hive performance benchmarks. http://issues.apache.org/jira/browse/HIVE-396.
[7] LibProcess Homepage. http://www.eecs.berkeley.edu/~benh/libprocess.
[8] Linux 2.6.33 release notes. http://kernelnewbies.org/Linux_2_6_33.
[9] Linux containers (LXC) overview document. http://lxc.sourceforge.net/lxc.html.
[10] Personal communication with Dhruba Borthakur from Facebook.
[11] Personal communication with Owen O'Malley and Arun C. Murphy from the Yahoo! Hadoop team.
[12] RightScale blog. blog.rightscale.com/2010/04/01/benchmarking-load-balancers-in-the-cloud.
[13] Solaris Resource Management. http://docs.sun.com/app/docs/doc/817-1592.
[14] The Next Generation of Apache Hadoop MapReduce. http://developer.yahoo.com/blogs/hadoop/posts/2011/02/mapreduce-nextgen.
[15] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Supercomputing '90*, 1990.
[16] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier, and F. Magniette. Mpich-v2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *Supercomputing '03*, 2003.
[17] T. Condie, N. Conway, P. Alvaro, and J. M. Hellerstein. MapReduce online. In *NSDI '10*, May 2010.
[18] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[19] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proc. HPDC '10*, 2010.
[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995.
[21] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *NSDI*, 2011.
[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.
[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. Technical Report UCB/EECS-2010-87, UC Berkeley, May 2010.
[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, 2007.
[25] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, November 2009.
[26] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HOTOS*, May 2009.
[27] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proc. ACM symposium on Cloud computing*, SoCC '10, 2010.
[28] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.
[29] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *ExpCS '07*, 2007.
[30] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. In *NSDI*, 2011.
[31] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus open-source cloud-computing system. In *CCA '08*, 2008.
[32] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2:129–138, April 1999.
[33] G. Staples. TORQUE resource manager. In *Proc. Supercomputing '06*, 2006.
[34] I. Stoica, H. Zhang, and T. S. E. Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM '97*, pages 249–262, 1997.
[35] J. Stone. Tachyon ray tracing system. http://jedi.ks.uiuc.edu/~johns/raytracer.
[36] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: flexible proportional-share resource management. In *OSDI*, 1994.
[37] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *SOSP '09*, pages 247–260, 2009.
[38] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *EuroSys 10*, 2010.
[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proc. HotCloud '10*, 2010.
[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce performance in heterogeneous environments. In *Proc. OSDI '08*, 2008.
[41] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing*, 1992.
[42] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *AAIM*, pages 337–348. Springer-Verlag, 2008.

[5] Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

Sharing the Data Center Network

Alan Shieh[‡†], Srikanth Kandula[‡], Albert Greenberg[∨], Changhoon Kim[∨], Bikas Saha[⋆]
Microsoft Research[‡], Cornell University[†], Windows Azure[∨], Microsoft Bing[⋆]
ashieh@cs.cornell.edu {srikanth,albert,chakim,bikas}@microsoft.com

Abstract– While today's data centers are multiplexed across many non-cooperating applications, they lack effective means to share their network. Relying on TCP's congestion control, as we show from experiments in production data centers, opens up the network to denial of service attacks and performance interference. We present Seawall, a network bandwidth allocation scheme that divides network capacity based on an administrator-specified policy. Seawall computes and enforces allocations by tunneling traffic through congestion controlled, point to multipoint, edge to edge tunnels. The resulting allocations remain stable regardless of the number of flows, protocols, or destinations in the application's traffic mix. Unlike alternate proposals, Seawall easily supports dynamic policy changes and scales to the number of applications and churn of today's data centers. Through evaluation of a prototype, we show that Seawall adds little overhead and achieves strong performance isolation.

1. INTRODUCTION

Data centers are crucial to provide the large volumes of compute and storage resources needed by today's Internet businesses including web search, content distribution and social networking. To achieve cost efficiencies and on-demand scaling, cloud data centers [5, 28] are highly-multiplexed shared environments, with VMs and tasks from multiple tenants coexisting in the same cluster. Since these applications come from unrelated customers, they are largely uncoordinated and mutually untrusting. Thus, the potential for network performance interference and denial of service attacks is high, and so performance predictability remains a key concern [8] for customers evaluating a move to cloud datacenters.

While data centers provide many mechanisms to schedule local compute, memory, and disk resources [10, 15], existing mechanisms for apportioning network resources fall short. End host mechanisms such as TCP congestion control (or variants such as TFRC and DCCP) are widely deployed, scale to existing traffic loads, and, to a large extent, determine network sharing today via a notion of flow-based fairness. However, TCP does little to isolate tenants from one another: poorly-designed or malicious applications can consume network capacity, to the detriment of other applications, by opening more flows or using non-compliant protocol implementations that ignore congestion control. Thus, while resource allocation using TCP is scalable and achieves high network utilization, it does not provide robust performance isolation.

Switch and router mechanisms (e.g., CoS tags, Weighted Fair Queuing, reservations, QCN [29]) are better decoupled from tenant misbehavior. However, these features, inherited from enterprise networks and the Internet, are of limited use when applied to the demanding cloud data center environment, since they cannot keep up with the scale and the churn observed in datacenters (e.g., numbers of tenants, arrival rate of new VMs), can only obtain isolation at the cost of network utilization, or might require new hardware.

For a better solution, we propose Seawall, an edge based mechanism that lets administrators prescribe how their network is shared. Seawall works irrespective of traffic characteristics such as the number of flows, protocols or participants. Seawall provides a simple abstraction: given a *network weight* for each local entity that serves as a traffic source (VM, process, etc.), Seawall ensures that along all network links, the share of bandwidth obtained by the entity is proportional to its weight. To achieve efficiency, Seawall is work-conserving, proportionally redistributing unused shares to currently active sources.

Beyond simply improving security by mitigating DoS attacks from malicious tenants and generalizing existing *use-what-you-pay-for* provisioning models, per-entity weights also enable better control over infrastructure services. Data centers often mix latency- and throughput-sensitive tasks with background infrastructure services. For instance, customer-generated web traffic contends with the demands of VM deployment and migration tasks. Per-entity weights obviate the need to hand-craft every individual service.

Further, per-entity weights also enable better control over application-level goals. Network allocation decisions can have significant impact on end-to-end metrics such as completion time or throughput. For example, in a map-reduce cluster, a reduce task with a high fan-in can open up many more flows than map tasks sharing the same bottleneck. Flow-based fairness prioritizes high fan-in reduce tasks over other tasks, resulting in imbalanced progress that leaves CPU resources idle and degrades cluster throughput. By contrast, Seawall decouples network allocation from communications patterns.

Seawall achieves scalable resource allocation by reducing the network sharing problem to an instance of distributed congestion control. The ubiquity of TCP shows

that such algorithms can scale to large numbers of participants, adapt quickly to change, and can be implemented strictly at the edge. Though Seawall borrows from TCP, Seawall's architecture and control loop ensure robustness against tenant misbehavior. Seawall uses a shim layer at the sender that makes policy compliance mandatory by forcing all traffic into congestion-controlled tunnels. To prevent tenants from bypassing Seawall, the shim runs in the virtualization or platform network stack, where it is well-isolated from tenant code.

Simply enforcing a separate TCP-like tunnel to every destination would permit each source to achieve higher rate by communicating with more destinations. Since this does not achieve the desired policy based on per-entity weights, Seawall instead uses a novel control loop that combines feedback from multiple destinations.

Overall, we make three contributions. First, we identify problems and missed opportunities caused by poor network resource allocation. Second, we explore at length the tradeoffs in building network allocation mechanisms for cloud data centers. Finally, we design and implement an architecture and control loop that are robust against malicious, selfish, or buggy tenant behavior. We have built a prototype of Seawall as a Windows NDIS filter. From experiments in a large server cluster, we show that Seawall achieves proportional sharing of the network while remaining agnostic to tenant protocols and traffic patterns and protects against UDP- and TCP-based DoS attacks. Seawall provides these benefits while achieving line rate with low CPU overhead.

2. PROBLEMS WITH NETWORK SHARING IN DATACENTERS

To understand the problems with existing network allocation schemes, we examine two types of clusters that consist of several thousands of servers and are used in production. The first type is that of public infrastructure cloud services that rent virtual machines along with other shared services such as storage and load balancers. In these datacenters, clients can submit arbitrary VM images and choose which applications to run, who to talk to, how much traffic to send, when to send that traffic, and what protocols to use to exchange that traffic (TCP, UDP, # of flows). The second type is that of platform cloud services that support map-reduce workloads. Consider a map-reduce cluster that supports a search engine. It is used to analyze logs and improve query and advertisement relevance. Though this cluster is shared across many users and business groups, the execution platform (i.e., the job compiler and runtime) is proprietary code controlled by the datacenter provider.

Through case studies on these datacenters we observe how the network is shared today, the problems that arise from such sharing and the requirements for an improved sharing mechanism.

In all datacenters, the servers have multiple cores, multiple disks, and tens of GBs of RAM. The network is a tree like topology [26] with 20–40 servers in a rack and a small over-subscription factor on the upstream links of the racks.

2.1 Performance interference in infrastructure cloud services

Recent measurements demonstrate considerable variation in network performance metrics – medium instances in EC2 experience throughput that can vary by 66% [25, 43]. We conjecture, based on anecdotal evidence, that a primary reason for the variation is the inability to control the network traffic share of a VM.

Unlike CPU and memory, network usage is harder to control because it is a distributed resource. For example, consider the straw man where each VM's network share is statically limited to a portion of the host's NIC rate (the equivalent of assigning the VM a fixed number of cores or a static memory size). A tenant with many VMs can cumulatively send enough traffic to overflow the receiver, some network link en route to that host, or other network bottlenecks. Some recent work [33] shows how to co-locate a trojan VM with a target VM. Using this, a malicious tenant can degrade the network performance of targeted victims. Finally, a selfish client, by using variable numbers of flows, or higher rate UDP flows, can hog network bandwidth.

We note that out-of-band mechanisms to mitigate these problems exist. Commercial cloud providers employ a combination of such mechanisms. First, the provider can account for the network usage of tenants (and VMs) and quarantine or ban the misbehavers. Second, cloud providers might provide even less visibility into their clusters to make it harder for a malicious client to co-locate with target VMs. However, neither approach is foolproof. Selfish or malicious traffic can mimic legitimate traffic, making it hard to distinguish. Further, obfuscation schemes may not stop a determined adversary.

Our position, instead, is to get at the root of the problem. The reason existing solutions fail is that they primarily rely on TCP flows. But VMs are free to choose their number of flows, congestion control variant, and even whether they respond to congestion, allowing a small number of VMs to disproportionately impact the network. Hence, we seek alternative ways to share the network that are independent of the clients' traffic matrices and implementations.

2.2 Poorly-performing schedules in Cosmos

We shift focus to Cosmos [9], a dedicated internal cluster that supports map-reduce workloads. We obtained detailed logs over several days from a production cluster with thousands of servers that supports the Bing search engine. The logs document the begin and end times of

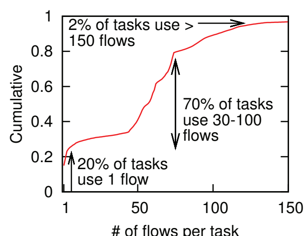


Figure 1: Distribution of the number of flows per task in Cosmos.

Task Type	#flows per task	% of net tasks
Aggregate	56.1	94.9
Partition	1.2	3.7
Extract	8.8	.2
Combine	2.3	1.0
other	1.0	.2

Figure 2: Variation in number of flows per task is due to the role of the task

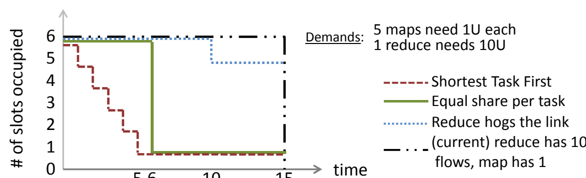


Figure 3: Poor sharing of the network leads to poor performance and wasted resources

jobs, tasks and flows in this cluster.

Performance interference happens here as well. Instances of high network load are common. A few entities (jobs, background services) contribute a substantial share of the traffic [22]. Tasks that move data over congested links suffer collateral damage – they are more likely to experience failures and become stragglers at the job level [6, 22].

Uniquely, however, we find that the de facto way of sharing the network leads to poor schedules. This is because schedulers for map-reduce platforms [27, 45] explicitly allocate local resources such as compute slots and memory. But, the underlying network primitives prevent them from exerting control over how tasks share the network. Map-reduce tasks naturally vary in the number of flows and the volume of data moved – a map task may have to read from just one location but a reduce task has to read data from all the map tasks in the preceding stage. Figure 1 shows that of the tasks that read data across racks, 20% of the tasks use just one flow, another 70% of the tasks vary between 30 and 100 flows, and 2% of the tasks use more than 150 flows. Figure 2 shows that this variation is due to the role of the task.

Because reduce tasks use a large number of flows, they starve other tasks that share the same paths. Even if the scheduler is tuned to assign a large number of compute slots for map tasks, just a few reduce tasks will cause these map tasks to be bottlenecked on the network. Thus, the compute slots held by the maps make little progress.

In principle, such unexpectedly idle slots could be put to better use on compute-heavy tasks or tasks that use less loaded network paths. However, current map-reduce schedulers do not support such load redistribution.[1]

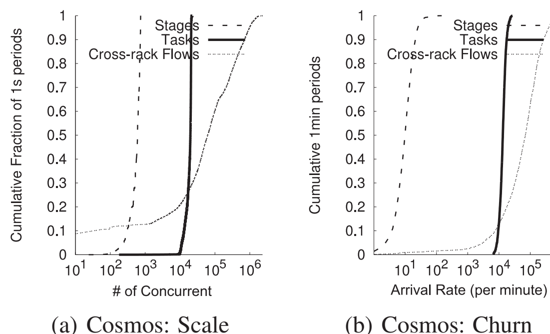
A simple example illustrates this problem. Figure 3 examines different ways of scheduling six tasks, five maps that each want to move 1 unit of data across a link of unit capacity and one reduce that wants to move 10 units of data from ten different locations over the same link. If the reduce uses 10 flows and each map uses 1 flow, as they do today, each of the flows obtains $\frac{1}{15}$'th of the link bandwidth and all six tasks finish at $t = 15$ (the schedule shown in black). The total activity period, since each task use local resources that no one else can use during the period it is active, is $6 * 15 = 90$.

If each task gets an even share of the link, it is easy to see that the map tasks will finish at $t = 6$ and the reduce task finishes at $t = 15$. In this case, the total activity period is $5 * 6 + 1 * 15 = 45$, or a 50% reduction in resource usage (the green solid line in Fig. 3). These spare resources can be used for other jobs or subsequent tasks within the same job.

The preceding example shows how the inherent variation in the way applications use the network can lead to poor schedules in the absence of control over how the network is shared. Our goal is to design ways of sharing the network that are efficient (no link goes idle if pent-up demand exists) and are independent of the traffic mix (UDP, #'s of TCP flows).

We note that prescribing the optimal bandwidth shares is a non-goal for this paper. In fact, evenly allocating bandwidth across tasks is not optimal for some metrics. If the provider has perfect knowledge about demands, scheduling the shortest remaining transfer first will minimize the activity period [18]. Going back to the example, this means that the five map tasks get exclusive access to the link and finish one after the other resulting in an activity period of 30 (the red dashed line in Fig. 3). However, this scheme has the side-effect of starving all the waiting transfers and requires perfect knowledge about client demands, which is hard to obtain in practice.

2.3 Magnitude of scale and churn



(a) Cosmos: Scale (b) Cosmos: Churn

Figure 4: Scale and churn seen in the observed datacenter.

We attempt to understand the nature of the sharing problem in production datacenters. We find that the number of classes to share bandwidth among is large and varies

frequently. Figure 4(a) shows the distribution of the number of concurrent entities that share the examined Cosmos cluster. Note that the x-axis is in log scale. We see that at median, there are 500 stages (e.g., map, reduce, join), 10^4 tasks and 10^5 flows in the cluster. The number of traffic classes required is at least two orders of magnitude larger than is feasible with current CoS tags or the number of WFQ/DRR queues that switches can handle per port.

Figure 4(b) shows the distribution of the number of new arrivals in the observed cluster. Note that the x-axis is again in log scale. At median, 10 new stages, 10^4 new tasks and $5 * 10^4$ new flows arrive in the cluster every minute. Anecdotal analysis of EC2, based on decoding the instance identifiers, concluded that O(10^4) new VM instances are requested each day [34]. Updating VLANs or re-configuring switches whenever a VM arrives is several orders of magnitude more frequent than is achievable in today's enterprise networks.

Each of the observed data centers is large, with up to tens of thousands of servers, thousands of ToR switches, several tens of aggregation switches, load balancers, etc. Predicting traffic is easier in platform datacenters (e.g., Cosmos) wherein high level descriptions of the jobs are available. However, the scale and churn numbers indicate that obtaining up-to-date information (e.g., within a minute) may be a practical challenge. In cloud datacenters (e.g., EC2) traffic is even harder to predict because customer's traffic is unconstrained and privacy concerns limit instrumentation.

3. REQUIREMENTS

From the above case studies and from interviews with operators of production clusters, we identify these requirements for sharing the datacenter network.

An ideal network sharing solution for datacenters has to scale, keep up with churn and retain high network utilization. It must do so without assuming well-behaved or TCP-compliant tenants. Since changes to the NICs and switches are expensive, take some time to standardize and deploy, and are hard to customize once deployed, edge- and software- based solutions are preferable.

- **Traffic Agnostic, Simple Service Interface:** Tenants cannot be expected to know or curtail the nature of their traffic. It is good business sense to accommodate diverse applications. While it is tempting to design sharing mechanisms that require tenants to specify a traffic matrix, i.e., the pattern and volume of traffic between the tenant's VMs, we find this to be an unrealistic burden. Changes in demands from the tenant's customers and dynamics of their workload (e.g., map-reduce) will change the requirements. Hence, it is preferable to keep a thin service interface, e.g., have tenants choose a class of network service.
- **Require no changes to network topology or hardware:** Recently, many data center network topologies have been proposed [2, 3, 16, 21]. Cost benefit tradeoffs indicate that the choice of topology depends on the intended usage. For example, EC2 recently introduced a full bisection bandwidth network for high performance computing (HPC); less expensive EC2 service levels continue to use the over-subscribed tree topology. To be widely applicable, mechanisms to share the network should be agnostic to network topology.
- **Scale to large numbers of tenants and high churn:** To have practical benefit, any network sharing mechanism would need to scale to support the large workloads seen in real datacenters.
- **Enforce sharing without sacrificing efficiency:** Statically apportioning fractions of the bandwidth improves sharing at the cost of efficiency and can result in bandwidth fragmentation that makes it harder to accommodate new tenants. At the same time, a tenant with pent up demand can use no more than its reservation even if the network is idle.

To meet these requirements, Seawall relies on *congestion-controlled tunnels* implemented in the host but requires no per-flow state within switches. In this way, Seawall is independent of the physical data center network. Seawall does benefit from measurements at switches, if they are available. Seawall scales to large numbers of tenants and handles high churn, because provisioning new VMs or tasks is entirely transparent to the physical network. As tenants, VMs, or tasks come and go, there is no change to the physical network through signaling or configuration. Seawall's design exploits the homogeneity of the data center environment, where end host software is easy to change and topology is predictable. These properties enable Seawall to use a system architecture and algorithms that are impractical on the Internet yet well-suited for data centers.

4. Seawall DESIGN

Seawall exposes the following abstraction. A *network weight* is associated with each entity that is sharing the network. The entity can be any traffic source that is confined to a single node, such as a VM, process, or collection of port numbers, but not a tenant or set of VMs. On each link in the network, Seawall provides the entity with a bandwidth share that is proportional to its weight; i.e., an entity k with weight w_k sending traffic over link l obtains this share of the total capacity of that link $Share(k,l) = \frac{w_k}{\Sigma_{i \in Active(l)} w_i}$. Here, $Active(l)$ is the set of entities actively sending traffic across l. The allocation is end-to-end, i.e., traffic to a destination will be limited by the smallest $Share(k,l)$ over links on the path to that destination. The allocation is also work-conserving: bandwidth that is unused because the entity needs less than its share or because its traffic is bottlenecked elsewhere is re-apportioned among other users of the link in

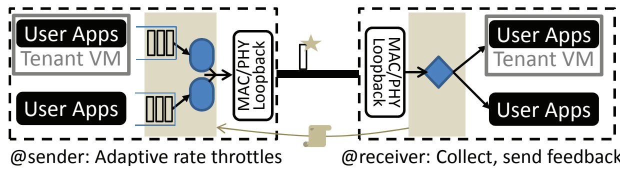


Figure 5: Seawall's division of functionality. (New components are shaded gray.)

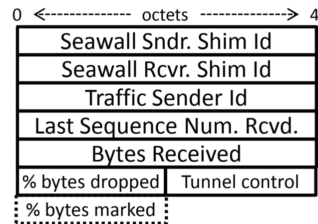


Figure 6: Content of Seawall's feedback packet

proportion to their weights. Here, we present a distributed technique that holds entities to these allocations while meeting our design requirements.

Weights can be adjusted dynamically and allocations re-converge rapidly. The special case of assigning the same weight to all entities divides bandwidth in a max-min fair fashion. By specifying equal weights to VMs, a public cloud provider can avoid performance interference from misbehaving or selfish VMs (§2.1). We defer describing further ways to configure weights and enforcing global allocations, such as over a set of VMs belonging to the same tenant, to §4.6.

4.1 Data path

To achieve the desired sharing of the network, Seawall sends traffic through congestion-controlled logical tunnels. As shown in Figure 5, these tunnels are implemented within a shim layer that intercepts all packets entering and leaving the server. At the sender, each tunnel is associated with an allowed rate for traffic on that tunnel, implemented as a rate limiter. The receive end of the tunnel monitors traffic and sends congestion feedback back to the sender. A bandwidth allocator corresponding to each entity uses feedback from all of the entity's tunnels to adapt the allowed rate on each tunnel. The bandwidth allocators take the network weights as parameters, work independently of each other, and together ensure that the network allocations converge to their desired values.

The Seawall shim layer is deployed to all servers in the data center by the management software that is responsible for provisioning and monitoring these servers (e.g., Autopilot, Azure Fabric). To ensure that only traffic controlled by Seawall enters the network, a provider can use attestation-based 802.1x authentication to disallow servers without the shim from connecting to the network.

The feedback to the control loop is returned at regular intervals, spaced T apart. It includes both explicit control signals from the receivers as well as congestion feedback about the path. Using the former, a receiver can explicitly block or rate-limit unwanted traffic. Using the latter, the bandwidth allocators adapt allowed rate on the tunnels. To help the receiver prepare congestion feedback, the shim at the sender maintains a byte sequence number per tunnel (i.e., per (sending entity, destination) pair). The sender shim stamps outgoing packets with the corresponding tunnel's current sequence number. The receiver detects losses in the same way as TCP, by looking for gaps in the received sequence number space. At the end of an interval, the receiver issues feedback that reports the number of bytes received and the percentage of bytes deemed to be lost (Figure 6). Optionally, if ECN is enabled along the network path, the feedback also relays the fraction of packets received with congestion marks.

We show efficient ways of stamping packets without adding a header and implementing queues and rate limiters in §5. Here, we describe the bandwidth allocator.

```
1:  .Begin (weight W)
2:  { rate r ← I, weight w ← W }                     ▷ Initialize
3:  .TakeFeedback (feedback f, proportion p)
4:  {
5:  if feedback f indicates loss then
6:      r ← r − r ∗ α ∗ p                ▷ Multiplicative Decrease
7:  else
8:      r ← r + w ∗ p                ▷ Weighted Additive Increase
9:  end if
10: }
```

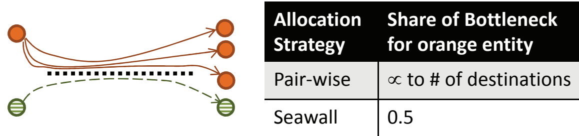
Class 1: A Strawman Bandwidth Allocator: an instance of this class is associated with each (entity, tunnel) pair.

4.2 Strawman

Consider the strawman bandwidth allocator in Class 1. Recall that the goal of the bandwidth allocator is to control the entity's network allocation as per the entity's network weight. Apart from the *proportion* variable, which we'll ignore for now, Class 1 is akin to weighted additive increase, multiplicative decrease (AIMD). It works as follows: when feedback indicates loss, it multiplicatively decreases the allowed rate by α. Otherwise, the rate increases by an additive constant.

This simple strawman satisfies some of our requirements. By making the additive increase step size a function of the entity's weight, the equilibrium rate achieved by an entity will be proportional to its weight. Unused shares are allocated to tunnels that have pent up demand, favoring efficiency over strict reservations. Global coordination is not needed. Further, when weights change, rates re-converge quickly (within one sawtooth period).

We derive the distributed control loop in Class 1 from TCP-Reno though any other flow-oriented protocol [4, 1, 29, 32] can be used, so long as it can extend to provide weighted allocations, as in MulTCP or MPAT [11, 39]. Distributed control loops are sensitive to variation in RTT. However, Seawall avoids this by using a constant feedback



Allocation Strategy	Share of Bottleneck for orange entity
Pair-wise	$\propto$ to # of destinations
Seawall	0.5

Figure 7: When entities talk to different numbers of destinations, pair-wise allocation of bandwidth is not sufficient. Reduce tasks behave like the orange entity while maps resemble the green. (Assume that both orange and green entities have the same weight.)

period T, chosen to be larger than the largest RTT of the intra datacenter paths controlled by Seawall. Conservatively, Seawall considers no feedback within a period of T as if a feedback indicating loss was received.

Simply applying AIMD, or any other distributed control loop, on a per-tunnel basis does not achieve the desired per-link bandwidth distribution. Suppose a tenant has N VMs and opens flows between every pair of VMs. This results in a tunnel between each VM; with one AIMD loop per tunnel, thus each VM achieves $O(N)$ times its allocation at the bottleneck link. Large tenants can overwhelm smaller tenants, as shown in Figure 7.

Seawall improves on this simple strawman in three ways. First, it has a unique technique to combine feedback from multiple destinations. By doing so, an entity's share of the network is governed by its network weight and is independent of the number of tunnels it uses (§4.3). The resulting policy is consistent with how cloud providers allocate other resources, such as compute and memory, to a tenant, yet is a significant departure from prior approaches to network scheduling. Second, the sawtooth behavior of AIMD leads to poor convergence on paths with high bandwidth-delay product. To mitigate this, Seawall modifies the adaptation logic to converge quickly and stay at equilibrium longer (§4.4). Third, we show how to nest traffic with different levels of responsiveness to congestion signals (e.g., TCP vs. UDP) within Seawall (§4.5).

4.3 Seawall's Bandwidth Allocator

The bandwidth allocator, associated with each entity, takes as input the network weight of that entity, the congestion feedback from all the receivers that the entity is communicating with and generates the allowed rate on each of the entity's tunnels. It has two parts: a distributed congestion control loop that computes the entity's cumulative share on each link and a local scheduler that divides that share among the various tunnels.

Step 1: Use distributed control loops to determine per-link, per-entity share. The ideal feedback would be per-link. It would include the cumulative usage of the entity across all the tunnels on this link, the total load on the link, and the network weights of all the entities using that link. Such feedback is possible if switches implement explicit feedback (e.g., XCP, QCN) or from programmable switch sampling (e.g., SideCar [38]). Lacking these, the baseline Seawall relies only on existing congestion signals such as end-to-end losses or ECN marks. These signals identify congested paths, rather than links.

To approximate link-level congestion information using path-level congestion signals, Seawall uses a heuristic based on the observation that a congested link causes losses in many tunnels using that link. The logic is described in Class 2. One instance of this class is associated with each entity and maintains separate per-link instances of the distributed control loop (rc_l). Assume for now that rc is implemented as per the strawman Class 1, though we will replace it with Class 3. The sender shim stores the feedback from each destination, and once every period T, applies all the feedback cumulatively (lines 8–10). The heuristic scales the impact of feedback from a given destination in proportion to the volume of traffic sent to that destination by the shim in the last period (line 7, 10).

To understand how this helps, consider the example in Figure 7. An instance of class 2, corresponding to the orange entity, cumulatively applies the feedback from all three destinations accessed via the bottleneck link to the single distributed control loop object representing that link. Since the proportions sum up to 1 across all destinations, the share of the orange entity will increase by only so much as that of the green entity.

A simplification follows because the shim at the receiver reports the fraction of bytes lost or marked. Hence, rather than invoking the distributed control loop once per destination, Class 2 computes just three numbers per link – the proportions of total feedback indicating loss, ECN marks, and neither, and invokes the distributed control loop once with each.

1: **.Begin** (weight W)
2: { rc_l.Begin(W) $\forall$ links l used by sender } $\triangleright$ Initialize
3: **.TakeFeedback** (feedback f_{dest})
4: { store feedback }
5: **.Periodically** ()
6: {
7: proportion of traffic to d, $p_d = \frac{f_d.bytesRcvd}{\sum f_i.bytesRcvd}$
8: **for all** destinations d **do**
9: **for all** links $l \in PathTo(d)$ **do**
10: rc_l.TakeFeedback(f_d, p_d)
11: **end for**
12: **end for**
13: $\triangleright$ rc_l now contains per-link share for this entity
14: $n_l \leftarrow$ count of dest with paths through link l
15: $\triangleright$ r_d is allowed rate to d
16: $r_d \leftarrow \min_{l \in PathTo(d)} \left(\left(\beta p_d + \frac{1-\beta}{n_l} \right) rc_l.rate \right)$
17: }

Class 2: Seawall's bandwidth allocator: A separate instance of this class is associated with each entity. It combines per-link distributed control loops (invoked in lines 2, 10) with a local scheduler (line 16).

Step 2: Convert per-link, per-entity shares to per-link, per-tunnel shares. Next, Seawall runs a local allocator to

assign rate limits to each tunnel that respects the entity's per-link rate constraints. A naïve approach divides each link's allowed rate evenly across all downstream destinations. For the example in Fig. 7, this leads to a $\frac{1}{3}'rd$ share of the bottleneck link to the three destinations of the orange entity. This leads to wasted bandwidth if the demands across destinations vary. For example, if the orange entity has demands $(2x, x, x)$ to the three destinations and the bottleneck's share for this entity is $4x$, dividing evenly causes the first destination to get no more than $\frac{4x}{3}$ while bandwidth goes wasted. Hence, Seawall apportions link bandwidth to destinations as shown in line 16, Class 2. The intuition is to adapt the allocations to match the demands. Seawall uses an exponential moving average that allocates β fraction of the link bandwidth proportional to current usage and the rest evenly across destinations. By default, we use $\beta = .9$. Revisiting the $(2x, x, x)$ example, note that while the first destination uses up all of its allowed share, the other two destinations do not, causing the first to get a larger share in the next period. In fact, the allowed share of the first destination converges to within 20% of its demand in four iterations.

Finally, Seawall converts these per-link, per-destination rate limits to a tunnel (i.e., per-path) rate limit by computing the minimum of the allowed rate on each link on the path. Note that Class 2 converges to a lower bound on the per-link allowed rate. At bottleneck links, this is tight. At other links, such as those used by the green flow in Figure 7 that are not the bottleneck, Class 2 can under-estimate their usable rate. Only when the green entity uses these other links on paths that do not overlap with the bottleneck, will the usable rate on those links increase. This behavior is the best that can be done using just path congestion signals and is harmless since the rate along each tunnel, computed as the minimum along each link on that path, is governed by the bottleneck.

4.4 Improving the Rate Adaptation Logic

Weighted AIMD suffers from inefficiencies as adaptation periods increase, especially for paths with high bandwidth-delay product [23] such as those in datacenters. Seawall uses control laws from CUBIC [32] to achieve faster convergence, longer dwell time at the equilibrium point, and higher utilization than AIMD. As with weighted AIMD, Seawall modifies the control laws to support weights and to incorporate feedback from multiple destinations. If switches support ECN, Seawall also incorporates the control laws from DCTCP [4] to further smooth out the sawtooth and reduce queue utilization at the bottleneck, resulting in reduced latency, less packet loss, and improved resistance against incast collapse.

The resulting control loop is shown in Class 3; the stability follows from that of CUBIC and DCTCP. Though we describe a rate-based variant, the equivalent window based versions are feasible and we defer those to future

```
1:  .Begin (weight W)
2:  { rate r ← I, weight w ← W, c ← 0, inc ← 0 }                ▷ Init
3:  .TakeFeedback (feedback f, proportion p)
4:  {
5:  c ← c + γ * p * (f.bytesMarked − c)
6:                          ▷ maintain smoothed estimate of congestion
7:  if f.bytesMarked > 0 then
8:      r_new ← r − r * α * p * c  ▷ Smoothed mult. decrease
9:      inc ← 0
10:     t_lastdrop ← now
11:     r_goal ← (r > r_goal)?r : (r+r_new)/2
12: else                                                ▷ Increase rate
13:     if r < r_goal then    ▷ Less than goal, concave increase
14:         Δt = min((now − t_lastdrop)/T_s, .9)
15:         Δr = δ * (r_goal − r) * (1 − Δt)^3
16:         r ← r + w * p * Δr
17:     else                          ▷ Above goal, convex increase
18:         r ← r + p * inc
19:         inc ← inc + w * p
20:     end if
21: end if
22: }
```

Class 3: Seawall**'s distributed control loop: an instance of this class is associated with each (link, entity) pair. Note that Class 2 invokes this loop (lines 2, 10).**

work. We elaborate on parameter choices in §4.6. Lines 14-17 cause the rate to increase along a concave curve, i.e., quickly initially and then slower as rate nears r_{goal}. After that, lines 18-19 implement convex increase to rapidly probe for a new rate. Line 5 maintains a smoothed estimate of congestion, allowing multiplicative decreases to be modulated accordingly (line 8) so that the average queue size at the bottleneck stays small.

4.5 Nesting Traffic Within Seawall

Nesting traffic of different types within Seawall's congestion-controlled tunnels leads to some special cases. If a sender always sends less than the rate allowed by Seawall, she may never see any loss causing her allowed rate to increase to infinity. This can happen if her flows are low rate (e.g., web traffic) or are limited by send or receive windows (flow control). Such a sender can launch a short overwhelming burst of traffic. Hence, Seawall clamps the rate allowed to a sender to a multiple of the largest rate she has used in the recent past. Clamping rates is common in many control loops, such as XCP [23], for similar reasons. The specific choice of clamp value does not matter as long as it is larger than the largest possible bandwidth increase during a Seawall change period.

UDP and TCP flows behave differently under Seawall. While a full burst UDP flow immediately uses all the rate that a Seawall tunnel allows, a set of TCP flows can take several RTTs to ramp up; the more flows, the faster the ramp-up. Slower ramp up results in lower shares on average. Hence, Seawall modifies the network stack to defer congestion control to Seawall's shim layer. All other

TCP functionality, such as flow control, loss recovery and in order delivery remain as before.

The mechanics of re-factoring are similar to Congestion Manager (CM) [7]. Each TCP flow queries the appropriate rate limiter in the shim (e.g., using shared memory) to see whether a send is allowed. Flows that have a backlog of packets register callbacks with the shim to be notified when they can next send a packet. In virtualized settings, the TCP stack defers congestion control to the shim by expanding the paravirtualized NIC interface. Even for tenants that bring their own OSes, the performance gain from refactoring the stack incentivizes adoption. Some recent advances in designing device drivers [36] reduce the overhead of signaling across the VM boundary. However, Seawall uses this simplification that requires less signaling: using hypervisor IPCs, the shim periodically reports a maximum congestion window to each VM to use for all its flows. The max congestion window is chosen large enough that each VM will pass packets to the shim yet small enough to not overflow the queues in front of the rate limiters in the shim.

We believe that deferring congestion control to the Seawall shim is necessary in the datacenter context. Enforcing network shares at the granularity of a flow no longer suffices (see §2). Though similar in spirit to Congestion Manager, Seawall refactors congestion control for different purposes. While CM does so to share congestion information among flows sharing a path, Seawall uses it to ensure that the network allocation policy holds regardless of the traffic mix. In addition, this approach allows for transparent changes to the datacenter transport.

4.6 Discussion

Here, we discuss details deferred from the preceding description of Seawall.

Handling WAN traffic: Traffic entering and leaving the datacenter is subject to more stringent DoS scrubbing at pre-defined chokepoints and, because WAN bandwidth is a scarce resource, is carefully rate-limited, metered and billed. We do not expect Seawall to be used for such traffic. However, if required, edge elements in the datacenter, such as load balancers or gateways, can funnel all incoming traffic into Seawall tunnels; the traffic then traverses a shim within the edge element. Traffic leaving the data center is handled analogously.

Mapping paths to links: To run Seawall, each sender requires path-to-link mapping for the paths that it is sending traffic on (line 10, Class 2). A sender can acquire this information independently, for example via a few traceroutes. In practice, however, this is much easier. Data center networks are automatically managed by software that monitors and pushes images, software and configuration to every node [19, 28]. Topology changes (e.g., due to failures and reconfiguration) are rare and can be disseminated automatically by these systems. Many pieces of today's datacenter ecosystem use topology information (e.g., Map-Reduce schedulers [27] and VM placement algorithms). Note that Seawall does work with a partial mapping (e.g., a high level mapping of each server to its rack, container, VLAN and aggregation switch) and does not need to identify bottleneck links. However, path-to-link mapping is a key enabler; it lets Seawall run over any datacenter network topology.

Choosing network weights: Seawall provides several ways to define the sending entity and the corresponding network weight. The precise choices depend on the datacenter type and application. When VMs are spun up in a cloud datacenter, the fabric sets the network weight of that VM alongside weights for CPU and memory. The fabric can change the VMs weight, if necessary, and Seawall re-converges rapidly. However, a VM cannot change its own weight. The administrator of a cloud datacenter can assign equal weights to all VMs, thereby avoiding performance interference, or assign weights in proportion to the size or price of the VM.

In contrast, the administrator of a platform datacenter can empower trusted applications to adjust their weights at run-time (e.g., via setsockopt()). Here, Seawall can also be used to specify weights per executable (e.g., background block replicator) or per process or per port ranges. The choice of weights could be based on information that the cluster schedulers have. For example, a map-reduce scheduler can assign the weight of each sender feeding a task in inverse proportion to the aggregation fan-in of that task, which he knows before hand. This ensures that each task obtains the same bandwidth (§2.2). Similarly, the scheduler can boost the weight of outlier tasks that are starved or are blocking many other tasks [6], thereby improving job completion times.

Enforcing global allocations: Seawall has so far focused on enforcing the network share of a local entity (VM, task etc.). This is complementary to prior work on Distributed Rate Limiters (DRL) [31] that controls the aggregate rate achieved by a collection of entities. Controlling just the aggregate rate is vulnerable to DoS: a tenant might focus the traffic of all of its VMs on a shared service (such as storage) or link (e.g., ToR containing victim tenant's servers), thereby interfering with the performance of other tenants while remaining under its global bandwidth cap. Combining Seawall with a global allocator such as DRL is simple. The Seawall shim reports each entity's usage to the global controller in DRL, which employs its global policy on the collection of entities and determines what each entity is allowed to send. The shim then caps the rate allowed to that entity to the minimum of the rate allowed by Seawall and the rate allowed by DRL's global policy. Further, the combination lets DRL scale better, since with Seawall, DRL need only track per-entity usage and not

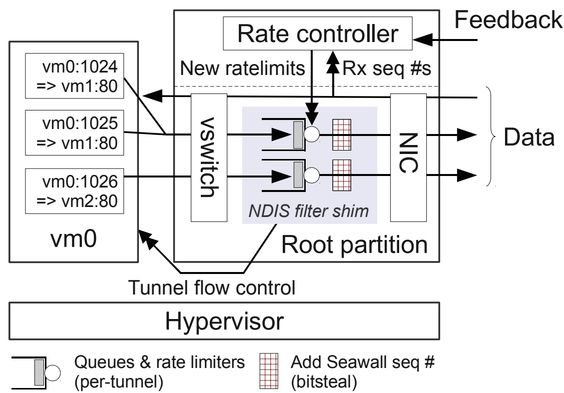


Figure 8: The Seawall prototype is split into an in-kernel NDIS filter shim (shaded gray), which implements the rate limiting datapath, and a userspace rate adapter, which implements the control loop. Configuration shown is for infrastructure data centers.

per-flow state that it would otherwise have to.

Choosing parameters: Whenever we adapt past work, we follow their guidance for parameters. Of the parameters unique to Seawall, their specific values have the following impact. We defer a formal analysis to future work. Reducing the feedback period T makes Seawall's adaptation logic more responsive at the cost of more overhead. We recommend choosing $T \in [10, 50]\ ms$. The multiplicative factor α controls the decrease rate. With the CUBIC/DCTCP control loop (see Class 3), Seawall is less sensitive to α than the AIMD control loop, since the former ramps back up more aggressively. In Class 2, β controls how much link rate is apportioned evenly versus based on current usage. With a larger β, the control loop reacts more quickly to changing demands but delays apportioning unused rate to destinations that need it. We recommend $\beta > .8$.

5. Seawall PROTOTYPE

The shim layer of our prototype is built as an NDIS packet filter (Figure 8). It interposes new code between the TCP/IP stack and the NIC driver. In virtualized settings, the shim augments the vswitch in the root partition. Our prototype is compatible with deployments that use the Windows 7 kernel as the server OS or as the root partition of Hyper-V. The shim can be adapted to other OSes and virtualization technologies, e.g., to support Linux and Xen, one can reimplement it as a Linux network queuing discipline module. For ease of experimentation, the logic to adapt rates is built in user space whereas the filters on the send side and the packet processing on the receive side are implemented in kernel space.

Clocking rate limiters: The prototype uses software-based token bucket filters to limit the rate of each tunnel. Implementing software rate limiters that work correctly and efficiently at high rates (e.g., 100s of Mbps) requires high precision interrupts; which are not widely available to drivers. Instead, we built a simple high precision clock. One core, per rack of servers, stays in a busy loop, and broadcasts a UDP heartbeat packet with the current time to all the servers within that rack once every 0.1ms; the shim layers use these packets to clock their rate limiters. We built a roughly equivalent window-based version of the Seawall shim as proof-of-concept. Windowing is easier to engineer, since it is self-clocking and does not require high precision timers, but incurs the expense of more frequent feedback packets (e.g., once every 10 packets).

Bit-stealing and stateless offload compatibility: A practical concern is the need to be compatible with NIC offloads. In particular, adding an extra packet header to support Seawall prevents the use of widely-used NIC offloads, such as large send offload (LSO) and receive side coalescing (RSC) which only work for known packet formats such as UDP or TCP. This leads to increased CPU overhead and decreased throughput. On a quad core 2.66 Intel Core2 Duo with an Intel 82567LM NIC, sending at the line rate of 1Gbps requires 20% more CPU without LSO (net: 30% without vs 10% with LSO) [37].

NIC vendors have plans to improve offload support for generic headers. To be immediately deployable without performance degradation, Seawall *steals* bits from existing packet headers, that is, it encodes information in parts of the packet that are unused or predictable and hence can be restored by the shim at the receiver. For both UDP and TCP, Seawall uses up to 16 bits from the IP ID field, reserving the lower order bits for the segmentation hardware if needed. For TCP packets, Seawall repurposes the timestamp option: it compresses the option Kind and Length fields from 16 bits down to 1 bit, leaving the rest for Seawall data. In virtualized environments, guest OSes are para-virtualized to always include timestamp options. The feedback is sent out-of-band in separate packets. We also found bit-stealing easier to engineer than adding extra headers, which could easily lead to performance degradation unless buffers were managed carefully.

Offloading rate limiters and direct I/O: A few emerging standards to improve network I/O performance, such as Direct I/O and SR-IOV, let guest VMs bypass the virtual switch and exchange packets directly with the NIC. But, this also bypasses the Seawall shim. Below, we propose a few ways to restore compatibility. However, we note that the loss of the security and manageability features provided by the software virtual switch has limited the deployment of direct I/O NICs in public clouds. To encourage deployment, vendors of such NICs plan to support new features specific to datacenters.

By offloading token bucket- and window-based limiters from the virtual switch to NIC or switch hardware, tenant traffic can be controlled even if guest VMs directly send packets to the hardware. To support Seawall, such offloaded rate limiters need to provide the same

granularity of flow classification (entity to entity tunnels) as the shim and report usage and congestion statistics. High end NICs that support stateful TCP, iSCSI, and RDMA offloads already support tens of thousands to millions of window-control engines in hardware. Since most such NICs are programmable, they can likely support the changes needed to return statistics to Seawall. Switch *policers* have similar scale and expressiveness properties. In addition, low cost programmable switches can be used to monitor the network for violations [38]. Given the diversity of implementation options, we believe that the design point occupied by Seawall, i.e., using rate- or window-controllers at the network edge, is feasible now and as data rates scale up.

6. EVALUATION

We ran a series of experiments using our prototype to show that Seawall achieves line rate with minimal CPU overhead, scales to typical data centers, converges to network allocations that are agnostic to communications pattern (i.e., number of flows and destinations) and protocol mix (i.e., UDP and TCP), and provides performance isolation. Through experiments with web workloads, we also demonstrate how Seawall can protect cloud-hosted services against DoS attacks, even those using UDP floods.

All experiments used the token bucket filter-based shim (i.e., rate limiter), which is our best-performing prototype and matches commonly-available hardware rate limiters. The following hold unless otherwise stated: (1) Seawall was configured with the default parameters specified in §4, (2) all results were aggregated from 10 two minute runs, with each datapoint a 15 second average and error bars indicating the 95% confidence interval.

Testbed: For our experiments, we used a 60 server cluster spread over three racks with 20 servers per rack. The physical machines were equipped with Xeon L5520 2.27 GHz CPUs (quad core, two hyperthreads per core), Intel 82576 NICs, and 4GB of RAM. The NIC access links were 1Gb/s and the links from the ToR switches up to the aggregation switch were 10Gb/s. There was no over-subscription within each rack. The ToR uplinks were 1:4 over-subscribed. We chose this topology because it is representative of typical data centers.

For virtualization, we use Windows Server 2008R2 Hyper-V with Server 2008R2 VMs. This version of Hyper-V exploits the Nehalem virtualization optimizations, but does not use the direct I/O functionality on the NICs. Each guest VM was provisioned with 1.5 GB of RAM and 4 virtual CPUs.

6.1 Microbenchmarks

6.1.1 *Throughput and overhead*

To evaluate the performance and overhead of Seawall, we measured the throughput and CPU overhead of tunneling a TCP connection between two machines through the shim. To minimize extraneous sources of noise, no other traffic was present in the testbed during each experiment and the sender and receiver transferred data from and to memory.

	Throughput (Mb/s)	**CPU @ Sender** (%)	**CPU @ Receiver** (%)
Seawall	947 ± 9	20.7 ± 0.6	14.2 ± 0.4
NDIS	977 ± 4	18.7 ± 0.4	13.5 ± 1.1
Baseline	979 ± 6	16.9 ± 1.9	10.8 ± 0.8

Table 1: CPU overhead comparison of Seawall, a null NDIS driver, and an unmodified network stack. Seawall achieved line rate with low overhead.

Seawall achieved nearly line rate at steady state, with negligible increase in CPU utilization, adding 3.8% at the sender and 3.4% at the receiver (Table 1). Much of this overhead was due to the overhead from installing a NDIS filter driver: the null NDIS filter by itself added 1.8% and 2.7% overhead, respectively. The NDIS framework is fairly light weight since it runs in the kernel and requires no protection domain transfers.

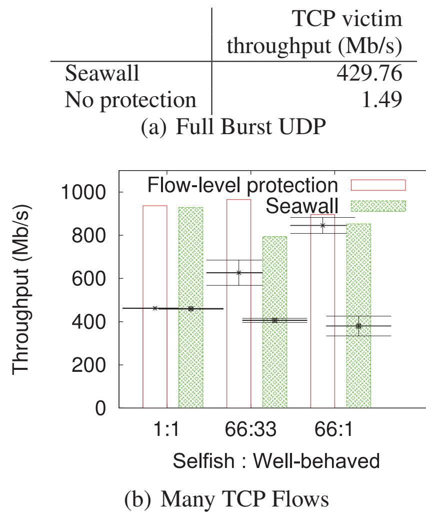
Subtracting out the contributions from the NDIS filter driver reveals the overheads due to Seawall: it incurred slightly more overhead on the sender than the receiver. This is expected since the sender does more work: on receiving packets, a Seawall receiver need only buffer congestion information and bounce it back to the sender, while the sender incurs the overhead of rate limiting and may have to merge congestion information from many destinations.

Seawall easily scales to today's data centers. The shim at each node maintains a rate limiter, with a few KBs of state each, for every pair of communicating entities terminating at that node. The per-packet cost on the data path is fixed regardless of data center size. A naive implementation of the rate controller incurs O(DL) complexity per sending entity (VM or task) where D is the number of destinations the VM communicates with and L is the number of links on paths to those destinations. In typical data center topologies, the diameter is small, and serves as an upper bound for L. All network stacks on a given node have collective state and processing overheads that grow at least linearly with D; these dominate the corresponding contributions from the rate controller and shim.

6.1.2 *Traffic-agnostic network allocation*

Seawall seeks to control the network share obtained by a sender, regardless of traffic. In particular, a sender should not be able to attain bandwidth beyond that allowed by the configured weight, no matter how it varies protocol type, number of flows, and number of destinations.

To evaluate the effectiveness of Seawall in achieving this goal, we set up the following experiment. Two physical nodes, hosting one VM each, served as the sources, with one VM dedicated to selfish traffic and the other to well-behaved traffic. One physical node served as the sink for



	TCP victim throughput (Mb/s)
Seawall	429.76
No protection	1.49

(a) Full Burst UDP

(b) Many TCP Flows

Figure 9: Seawall **ensures that despite using full burst UDP flows or many TCP flows, the share of a selfish user is held proportional to its weight. (In (b), the bars show total throughput, with the fraction below the divider corresponding to selfish traffic and the fraction above corresponding to well-behaved traffic.)**

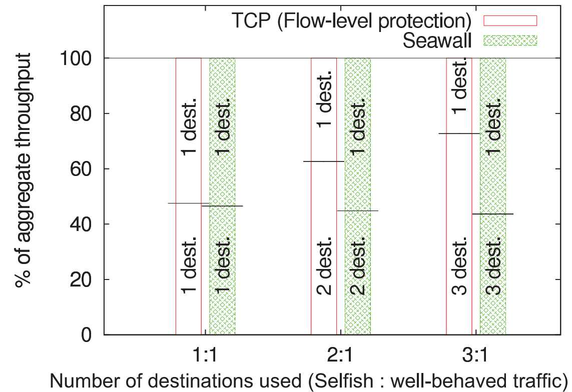


Figure 10: By combining feedback from multiple destinations, Seawall **ensures that the share of a sender remains independent of the number of destinations it communicates with. (The fraction of the bar below the divider corresponds to the fraction of bottleneck throughput achieved by selfish traffic.)**

all traffic; it was configured with two VMs, with one VM serving as the sink for well-behaved traffic and the other serving as the sink for selfish traffic.

Both well-behaved and selfish traffic used the same number of source VMs, with all Seawall senders assigned the same network weight. The well-behaved traffic consisted of a single long-lived TCP flow from each source, while the selfish traffic used one of three strategies to achieve a higher bandwidth share: using full burst UDP flow, using large numbers of TCP flows, and using many destinations

Selfish traffic = Full-burst UDP: Figure 9(a) shows the aggregate bandwidth achieved by the well-behaved traffic (long-lived TCP) when the selfish traffic consisted of full rate UDP flows. The sinks for well-behaved and selfish traffic were colocated on a node with a single 1Gbps NIC. Because each sender had equal weight, Seawall assigned half of this capacity to each sender. Without Seawall, selfish traffic overwhelms well-behaved traffic, leading to negligible throughput for well-behaved traffic. By bundling the UDP traffic inside a tunnel that imposed congestion control, Seawall ensured that well-behaved traffic retained reasonable performance.

Selfish traffic = Many TCP flows: Figure 9(b) shows the bandwidth shares achieved by selfish and well-behaved traffic when selfish senders used many TCP flows. As before, well-behaved traffic ideally should have achieved $\frac{1}{2}$ of the bandwidth. When selfish senders used the same number of flows as well-behaved traffic, bandwidth was divided evenly (left pair of bars). In runs without Seawall, selfish senders that used twice as many flows obtained $\frac{2}{3}$'rds the bandwidth because TCP congestion control divided bandwidth evenly across flows (middle pair of bars). Runs with Seawall resulted in approximately even bandwidth allocation. Note that Seawall achieved slightly lower throughput in aggregate. This was due to slower recovery after loss– the normal traffic had one sawtooth per TCP flow whereas Seawall had one per source VM; we believe this can be improved using techniques from §4. When the selfish traffic used 66 times more flows, it achieved a dominant share of bandwidth; the well-behaved traffic was allocated almost no bandwidth (rightmost pair of bars). We see that despite the wide disparity in number of flows, Seawall divided bandwidth approximately evenly. Again, Seawall improved the throughput of well-behaved traffic (the portion above the divider) by several orders of magnitude.

Selfish traffic = Arbitrarily many destinations: This experiment evaluated Seawall's effectiveness against selfish tenants that opened connections to many destinations. The experiment used a topology similar to that in Figure 7. A well-behaved sender VM and a selfish sender VM were located on the same server. Each sink was a VM and ran on a separate, dedicated machine. The well-behaved traffic was assigned one sink machine and the selfish traffic was assigned a variable number of sink machines. Both well-behaved and selfish traffic consisted of one TCP flow per sink. As before, the sending VMs were configured with the same weight, so that well-behaved traffic would achieve an even share of the bottleneck.

Figure 10 plots the fraction of bottleneck bandwidth achieved by well-behaved traffic with and without Seawall. We see that without Seawall, the share of the selfish traffic was proportional to the number of destinations. With Seawall, the share of the well-behaved traffic remained constant at approximately half, independent of the number

	Throughput (Mb/s)	Latency (s)
Seawall	181	0.61
No protection	157	0.91

Figure 11: Despite bandwidth pressure, Seawall ensures that the average HTTP request latency remains small without losing throughput.

of destinations.

6.2 Performance isolation for web servers

To show that Seawall protects against performance interference similar to that shown in §2, we evaluated the achieved level of protection against a DoS attack on a web server. Since cloud datacenters are often used to host web-accessible services, this is a common use case.

In this experiment, an attacker targeted the HTTP responses sent from the web server to its clients. To launch such attacks, an adversary places a source VM and a sink VM such that traffic between these VMs crosses the same bottleneck links as the web server. The source VM is close to the server, say on the same rack or machine, while the sink VM is typically on another rack. Depending on where the sink is placed, the attack can target the ToR uplink or another link several hops away.

All machines were colocated on the same rack. The web server VM, running Microsoft IIS 7, and attacker source VM, generating UDP floods, resided in separate, dedicated physical machines. A single web client VM requested data from the server and shared a physical machine with an attacker sink VM. The web clients used WcAsync to generate well-formed web sessions. Session arrivals followed a Poisson process and were exponentially sized with a mean of 10 requests. Requests followed a WebStone distribution, varying in size from 500B responses to 5MB responses with smaller files being much more popular.

As expected, a full-rate UDP attack flood caused congestion on the access link of the web client, reducing throughput to close to zero and substantially increasing latency. With Seawall, the web server behaved as if there were no attack. To explore data points where the access link was not overwhelmed, we dialed down the UDP attack rate to 700Mbps, enough to congest the link but not to stomp out the web server's traffic. While achieving roughly the same throughput as in the case of no protection, Seawall improved the latency observed by web traffic by almost 50% (Figure 11). This is because sending the attack traffic through a congestion controlled tunnel ensured that the average queue size at the bottleneck stays small, thereby reducing queuing delays.

7. DISCUSSION

Here, we discuss how Seawall can be used to implement rich cloud service models that provide bandwidth guarantees to tenants, the implications of our architectural decisions given trends in data centers and hardware, and the benefits of jointly modifying senders and receivers to achieve new functionality in data center networks.

7.1 Sharing policies

Virtual Data Centers (VDCs) have been proposed [20, 17, 40] as a way to specify tenant networking requirements in cloud data centers. VDCs seek to approximate, in terms of security isolation and performance, a dedicated data center for each tenant and allows tenants to specify SLA constraints on network bandwidth at per-port and per-source/dest-pair granularities. When allocating tenant VMs to physical hardware, the data center fabric simultaneously satisfies the specified constraints while optimizing node and network utilization.

Though Seawall policies could be seen as a simpler-to-specify alternative to VDCs that closely matches the provisioning knobs (e.g., disk, CPU, and memory size) of current infrastructure clouds, Seawall's weight-based policies can enhance VDCs in several ways. Some customers, through analysis or operational experience, understand the traffic requirements of their VMs; VDCs are attractive since they can exploit such detailed knowledge to achieve predictable performance. To improve VDCs with Seawall, the fabric uses weights to implement the hard bandwidth guarantees specified in the SLA: with appropriate weights, statically chosen during node- and path-placement, Seawall will converge to the desired allocation. Unlike implementations based on static reservations [17], the Seawall implementation is work-conserving, max-min fair, and achieves higher utilization through statistical multiplexing.

Seawall also improves a tenant's control of its own VDC. Since Seawall readily accepts dynamic weight changes, each tenant can adjust its allocation policy at a fine granularity in response to changing application needs. The fabric permits tenants to reallocate weights between different tunnels so long as the resulting weight does not exceed the SLA; this prevents tenants from stealing service and avoids having to rerun the VM placement optimizer.

7.2 System architecture

Topology assumptions: The type of topology and available bandwidth affects the complexity requirements of network sharing systems. In full bisection bandwidth topologies, congestion can only occur at the core. System design is simplified [44, 40, 30], since fair shares can be computed solely from information about edge congestion, without any topology information or congestion feedback from the core.

Seawall supports general topologies, allowing it to provide benefits even in legacy or cost-constrained data centers networks. Such topologies are typically bandwidth-constrained in the core; all nodes using a given core link need to be accounted for to achieve fair sharing, bandwidth reservations, and congestion control. Seawall ex-

plicitly uses topology information in its control layer to prevent link over-utilization.

Rate limiters and control loops: Using more rate limiters enables a network allocation system to support richer, more granular policies. Not having enough rate limiters can result in aliasing. For instance, VM misbehavior can cause Gatekeeper [40] to penalize unrelated VMs sending to the same destination. Using more complex rate limiters can improve system performance. For instance, rate limiters based on multi-queue schedulers such as DWRR or Linux's hierarchical queuing classes can utilize the network more efficiently when rate limiter parameters and demand do not match, and the self-clocking nature of window-based limiters can reduce switch buffering requirements as compared to rate-based limiters. However, having a large number of complex limiters can constrain how a network sharing architecture can be realized, since NICs and switches do not currently support such rate limiters at scale.

To maximize performance and policy expressiveness, a network allocation system should support a large number of limiters of varying capability. The current Seawall architecture can support rate- and window-based limiters based in hardware and software. As future work, we are investigating ways to map topology information onto hierarchical limiters; to compile policies given a limited number of available hardware limiters; and to tradeoff rate limiter complexity with controller complexity, using longer adaptation intervals when more capable rate limiters are available.

7.3 Partitioning sender/receiver functionality

Control loops can benefit from receiver-side information and coordination, since the receiver is aware of the current traffic demand from all sources and can send feedback to each with lower overhead. Seawall currently uses a receiver-driven approach customized for map-reduce to achieve better network scheduling; as future work we are building a general solution at the shim layer.

In principal, a purely receiver-directed approach to implementing a new network allocation policy, such as that used in [44, 40], might reduce system complexity since the sender TCP stack does not need to be modified. However, virtualization stack complexity does not decrease substantially, since the rate controller simply moves from the sender to the receiver. Moreover, limiting changes to one endpoint in data centers provides little of the adoption cost advantages found in the heterogeneous Internet environment. Modifying the VMs to defer congestion control to other layers can help researchers and practitioners to identify and deploy new network sharing policies and transport protocols for the data center.

A receiver-only approach can also *add* complexity. While some allocation policies are easy to attain by treating the sender as a black box, others are not. For instance, eliminating fatesharing from Gatekeeper and adding weighted, fair work-conserving scheduling appears non-trivial. Moreover, protecting a receiver-only approach from attack requires adding a detector for non-conformant senders. While such detectors have been studied for WAN traffic [13], it is unclear whether they are feasible in the data center. Such detectors might also permit harmful traffic that running new, trusted sender-side code can trivially exclude.

8. RELATED WORK

Proportional allocation of shared resources has been a recurring theme in the architecture and virtualization communities [42, 15]. To the best of our knowledge, Seawall is the first to extend this to the data center network and support generic sending entities (VMs, applications, tasks, processes, etc.).

Multicast congestion control [14], while similar at first blush, targets a very different problem since they have to allow for any participant to send traffic to the group while ensuring TCP-friendliness. It is unclear how to adapt these schemes to proportionally divide the network.

Recent work in hypervisor, network stack, and software routers have shown that software-based network processing, like that used in Seawall for monitoring and rate limiting, can be more flexible than hardware-based approaches yet achieve high performance. [35] presents an optimized virtualization stack that achieves comparable performance to direct I/O. The Sun Crossbow network stack provides an arbitrary number of bandwidth-limited virtual NICs [41]. Crossbow provides identical semantics regardless of underlying physical NIC and transparently leverages offloads to improve performance. Seawall's usage of rate limiters can benefit from these ideas.

QCN is an emerging Ethernet standard for congestion control in datacenter networks [29]. In QCN, upon detecting a congested link, the switch sends feedback to the heavy senders. The feedback packet uniquely identifies the flow and congestion location, enabling senders that receive feedback to rate limit specific flows. QCN uses explicit feedback to drive a more aggressive control loop than TCP. While QCN can throttle the heavy senders, it is not designed to provide fairness guarantees, tunable or otherwise. Further, QCN requires changes to switch hardware and can only cover purely Layer 2 topologies.

Much work has gone into fair queuing mechanisms in switches [12]. Link local sharing mechanisms, such as Weighted Fair Queuing and Deficit Round Robin, separate traffic into multiple queues at each switch port and arbitrate service between the queues in some priority or proportion. NetShare [24] builds on top of WFQ support in switches. This approach is useful to share the network between a small number of large sending entities (e.g., a whole service type, such as "Search" or "Distributed storage" in a platform data center). The number of queues

available in today's switches, however, is several orders of magnitude smaller than the numbers of VMs and tasks in today's datacenters. More fundamentally, since link local mechanisms lack end-to-end information they can let significant traffic through only to be dropped at some later bottleneck on the path. Seawall can achieve better scalability by mapping many VMs onto a small, fixed number of queues and achieves better efficiency by using end-to-end congestion control.

9. FINAL REMARKS

Economies of scale are pushing distributed applications to co-exist with each other on shared infrastructure. The lack of mechanisms to apportion network bandwidth across these entities leads to a host of problems, from reduced security to unpredictable performance and to poor ability to improve high level objectives such as job completion time. Seawall is a first step towards providing data center administrators with tools to divide their network across the sharing entities without requiring any cooperation from the entities. It is novel in its ability to scale to massive numbers of sharing entities and uniquely adapts ideas from congestion control to the problem of enforcing network share agnostic to traffic type. The design space that Seawall occupies – push functionality to software at the network edge – appears well-suited to emerging hardware trends in data center and virtualization hardware.

Acknowledgements

We thank Deepak Bansal, Dave Maltz, our shepherd Bill Weihl and the NSDI reviewers for discussions that improved this work.

Notes

[1] Perhaps because it is hard to predict such events and find appropriate tasks at short notice. Also, running more tasks requires spare memory and has initialization overhead.

References

[1] *Understanding the Available Bit Rate (ABR) Service Category for ATM VCs*. Cisco Systems, 2006.

[2] A. Greenberg, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[4] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, 2010.

[5] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[6] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.

[7] H. Balakrishnan, H. Rahul, and S. Seshan. An integrated congestion management architecture for internet hosts. In *SIGCOMM*, 1999.

[8] Bill Claybrook. Comparing cloud risks and virtualization risks for data center apps. http://searchdatacenter.techtarget.com/tip/0,289483,sid80_gci1380652,00.html.

[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[10] L. Cherkasova, D. Gupta, and A. Vahdat. Comparison of the three CPU schedulers in Xen. In *SIGMETRICS*, 2007.

[11] J. Crowcroft and P. Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM CCR*, 28(3), 1998.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *ACM CCR*, 19(4), 1989.

[13] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE TON*, 7(4), 1999.

[14] J. Golestani and K. Sabnani. Fundamental observations on multicast congestion control in the Internet. In *INFOCOM*, 1999.

[15] A. Gulati and C. A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *FAST*, 2009.

[16] C. Guo et al. Bcube: High performance, server-centric network architecture for data centers. In *SIGCOMM*, 2009.

[17] C. Guo, G. Lu, H. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *ACM CoNEXT*, 2010.

[18] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM TOCS*, 21(2), 2003.

[19] M. Isard. Autopilot: Automatic Data Center Management. *OSR*, 41(2), 2007.

[20] M. Kallahalla, M. Yusal, R. Swaminathan, D. Lowell, M. Wray, T. Christian, N. Edwards, C. I. Dalton, and F. Gittler. SoftUDC: A Software-Based Data Center for Utility Computing. *Computer*, 2004.

[21] S. Kandula, J. Padhye, and P. Bahl. Flyways to de-congest data center networks. In *HotNets*, 2009.

[22] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements & Analysis. In *IMC*, 2009.

[23] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[24] T. Lam et al. NetShare : Virtualizing Data Center Networks across Services. Technical Report CS2010-0957, UCSD, 2010.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: Comparing public cloud providers. In *IMC*, 2010.

[26] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[27] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP*, 2009.

[28] Microsoft. An Overview of Windows Azure. http://download.microsoft.com/download/A/A/6/AA6A260A-B920-4BBC-AE33-8815996CD8FB/02-Article Introduction to Windows Azure.docx.

[29] R. Pan, B. Prabhakar, and A. Laxmikantha. QCN: Quantized Congestion Notification. http://www.ieee802.org/1/files/public/docs2007/au-prabhakar-qcn-description.pdf, 2007.

[30] L. Popa, S. Y. Ko, and S. Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *HotNets*, 2010.

[31] B. Raghavan, K. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren. Cloud control with distributed rate limiting. In *SIGCOMM*, 2007.

[32] I. Rhee and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *OSR*, 42(5), 2008.

[33] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *ACM CCS*, 2009.

[34] G. Rosen. Anatomy of an Amazon EC2 Resource ID. http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/.

[35] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt. Bridging the Gap between Software and Hardware Techniques for I / O Virtualization. Technical Report HPL-2008-39, HP Labs, 2008.

[36] L. Shalev, J. Satran, E. Borovik, and M. Ben-yehuda. IsoStack Highly Efficient Network Processing on Dedicated Cores. In *USENIX ATC*.

[37] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance isolation for cloud datacenter networks. In *HotCloud*, 2010.

[38] A. Shieh, S. Kandula, and E. Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[39] M. Singh, P. Pradhan, and P. Francis. MPAT: Aggregate TCP Congestion Management as a Building Block for Internet QoS. In *ICNP*, 2004.

[40] P. V. Soares, J. R. Santos, N. Tolia, D. Guedes, and Y. Turner. Gatekeeper: Distributed Rate Control for Virtualized Datacenters. Technical Report HPL-2010-151, HP Labs, 2010.

[41] S. Tripathi, N. Droux, T. Srinivasan, and K. Belgaied. Crossbow: from hardware virtualized NICs to virtualized networks. In *ACM VISA*, 2009.

[42] C. A. Waldspurger. *Lottery and Stride Scheduling : Proportional Share Resource Management*. PhD thesis, MIT, 1995.

[43] G. Wang and T. S. E. Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[44] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *ACM CoNEXT*, 2010.

[45] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

Dominant Resource Fairness: Fair Allocation of Multiple Resource Types

Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, Ion Stoica
University of California, Berkeley
{alig,matei,benh,andyk,shenker,istoica}@cs.berkeley.edu

Abstract

We consider the problem of fair resource allocation in a system containing different resource types, where each user may have different demands for each resource. To address this problem, we propose *Dominant Resource Fairness (DRF)*, a generalization of max-min fairness to multiple resource types. We show that DRF, unlike other possible policies, satisfies several highly desirable properties. First, DRF incentivizes users to share resources, by ensuring that no user is better off if resources are equally partitioned among them. Second, DRF is strategy-proof, as a user cannot increase her allocation by lying about her requirements. Third, DRF is envy-free, as no user would want to trade her allocation with that of another user. Finally, DRF allocations are Pareto efficient, as it is not possible to improve the allocation of a user without decreasing the allocation of another user. We have implemented DRF in the Mesos cluster resource manager, and show that it leads to better throughput and fairness than the slot-based fair sharing schemes in current cluster schedulers.

1 Introduction

Resource allocation is a key building block of any shared computer system. One of the most popular allocation policies proposed so far has been *max-min fairness*, which maximizes the minimum allocation received by a user in the system. Assuming each user has enough demand, this policy gives each user an equal share of the resources. Max-min fairness has been generalized to include the concept of weight, where each user receives a share of the resources proportional to its weight.

The attractiveness of weighted max-min fairness stems from its generality and its ability to provide performance isolation. The weighted max-min fairness model can support a variety of other resource allocation policies, including priority, reservation, and deadline based allocation [31]. In addition, weighted max-min fairness ensures isolation, in that a user is guaranteed to receive her share irrespective of the demand of the other users.

Given these features, it should come as no surprise that a large number of algorithms have been proposed to implement (weighted) max-min fairness with various degrees of accuracy, such as round-robin, proportional resource sharing [32], and weighted fair queueing [12]. These algorithms have been applied to a variety of resources, including link bandwidth [8, 12, 15, 24, 27, 29], CPU [11, 28, 31], memory [4, 31], and storage [5].

Despite the vast amount of work on fair allocation, the focus has so far been primarily on a *single* resource type. Even in multi-resource environments, where users have heterogeneous resource demands, allocation is typically done using a single resource abstraction. For example, fair schedulers for Hadoop and Dryad [1, 18, 34], two widely used cluster computing frameworks, allocate resources at the level of fixed-size partitions of the nodes, called *slots*. This is despite the fact that different jobs in these clusters can have widely different demands for CPU, memory, and I/O resources.

In this paper, we address the problem of fair allocation of multiple types of resources to users with heterogeneous demands. In particular, we propose Dominant Resource Fairness (DRF), a generalization of max-min fairness for multiple resources. The intuition behind DRF is that in a multi-resource environment, the allocation of a user should be determined by the user's *dominant share*, which is the maximum share that the user has been allocated of any resource. In a nutshell, DRF seeks to maximize the minimum dominant share across all users. For example, if user A runs CPU-heavy tasks and user B runs memory-heavy tasks, DRF attempts to equalize user A's share of CPUs with user B's share of memory. In the single resource case, DRF reduces to max-min fairness for that resource.

The strength of DRF lies in the properties it satisfies. These properties are trivially satisfied by max-min fairness for a single resource, but are non-trivial in the case of multiple resources. Four such properties are

sharing incentive, strategy-proofness, Pareto efficiency, and envy-freeness. DRF provides incentives for users to share resources by guaranteeing that no user is better off in a system in which resources are statically and equally partitioned among users. Furthermore, DRF is strategy-proof, as a user cannot get a better allocation by lying about her resource demands. DRF is Pareto-efficient as it allocates all available resources subject to satisfying the other properties, and without preempting existing allocations. Finally, DRF is envy-free, as no user prefers the allocation of another user. Other solutions violate at least one of the above properties. For example, the preferred [3, 22, 33] fair division mechanism in microeconomic theory, Competitive Equilibrium from Equal Incomes [30], is not strategy-proof.

We have implemented and evaluated DRF in Mesos [16], a resource manager over which multiple cluster computing frameworks, such as Hadoop and MPI, can run. We compare DRF with the slot-based fair sharing scheme used in Hadoop and Dryad and show that slot-based fair sharing can lead to poorer performance, unfairly punishing certain workloads, while providing weaker isolation guarantees.

While this paper focuses on resource allocation in datacenters, we believe that DRF is generally applicable to other multi-resource environments where users have heterogeneous demands, such as in multi-core machines.

The rest of this paper is organized as follows. Section 2 motivates the problem of multi-resource fairness. Section 3 lists fairness properties that we will consider in this paper. Section 4 introduces DRF. Section 5 presents alternative notions of fairness, while Section 6 analyzes the properties of DRF and other policies. Section 7 provides experimental results based on traces from a Facebook Hadoop cluster. We survey related work in Section 8 and conclude in Section 9.

2 Motivation

While previous work on weighted max-min fairness has focused on single resources, the advent of cloud computing and multi-core processors has increased the need for allocation policies for environments with multiple resources and heterogeneous user demands. By *multiple resources* we mean resources of different *types*, instead of multiple instances of the same interchangeable resource.

To motivate the need for multi-resource allocation, we plot the resource usage profiles of tasks in a 2000-node Hadoop cluster at Facebook over one month (October 2010) in Figure 1. The placement of a circle in Figure 1 indicates the memory and CPU resources consumed by tasks. The size of a circle is logarithmic to the number of tasks in the region of the circle. Though the majority of tasks are CPU-heavy, there exist tasks that are memory-

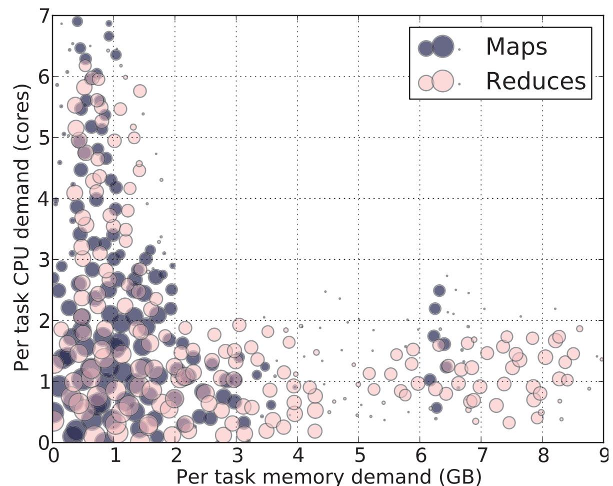


Figure 1: CPU and memory demands of tasks in a 2000-node Hadoop cluster at Facebook over one month (October 2010). Each bubble's size is logarithmic in the number of tasks in its region.

heavy as well, especially for reduce operations.

Existing fair schedulers for clusters, such as Quincy [18] and the Hadoop Fair Scheduler [2, 34], ignore the heterogeneity of user demands, and allocate resources at the granularity of slots, where a slot is a fixed fraction of a node. This leads to inefficient allocation as a slot is more often than not a poor match for the task demands.

Figure 2 quantifies the level of fairness and isolation provided by the Hadoop MapReduce fair scheduler [2, 34]. The figure shows the CDFs of the ratio between the task CPU demand and the slot CPU share, and of the ratio between the task memory demand and the slot memory share. We compute the slot memory and CPU shares by simply dividing the total amount of memory and CPUs by the number of slots. A ratio of 1 corresponds to a perfect match between the task demands and slot resources, a ratio below 1 corresponds to tasks underutilizing their slot resources, and a ratio above 1 corresponds to tasks over-utilizing their slot resources, which may lead to thrashing. Figure 2 shows that most of the tasks either underutilize or overutilize some of their slot resources. Modifying the number of slots per machine will not solve the problem as this may result either in a lower overall utilization or more tasks experiencing poor performance due to over-utilization (see Section 7).

3 Allocation Properties

We now turn our attention to designing a max-min fair allocation policy for multiple resources and heterogeneous requests. To illustrate the problem, consider a system consisting of 9 CPUs and 18 GB RAM, and two users: user A runs tasks that require $\langle 1 \text{ CPUs}, 4 \text{ GB}\rangle$ each, and user B runs tasks that require $\langle 3 \text{ CPUs}, 1 \text{ GB}\rangle$ each. What constitutes a fair allocation policy for this case?

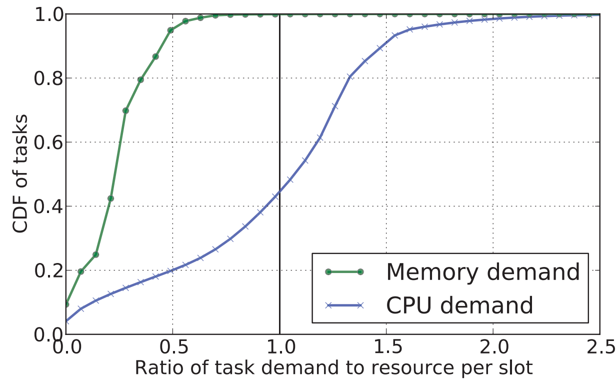


Figure 2: CDF of demand to slot ratio in a 2000-node cluster at Facebook over a one month period (October 2010). A demand to slot ratio of 2.0 represents a task that requires twice as much CPU (or memory) than the slot CPU (or memory) size.

One possibility would be to allocate each user half of every resource. Another possibility would be to equalize the aggregate (*i.e.,* CPU plus memory) allocations of each user. While it is relatively easy to come up with a variety of possible "fair" allocations, it is unclear how to evaluate and compare these allocations.

To address this challenge, we start with a set of desirable properties that we believe any resource allocation policy for multiple resources and heterogeneous demands should satisfy. We then let these properties guide the development of a fair allocation policy. We have found the following four properties to be important:

1. *Sharing incentive:* Each user should be better off sharing the cluster, than exclusively using her own partition of the cluster. Consider a cluster with identical nodes and n users. Then a user should not be able to allocate more tasks in a cluster partition consisting of $\frac{1}{n}$ of all resources.

2. *Strategy-proofness:* Users should not be able to benefit by lying about their resource demands. This provides incentive compatibility, as a user cannot improve her allocation by lying.

3. *Envy-freeness:* A user should not prefer the allocation of another user. This property embodies the notion of fairness [13, 30].

4. *Pareto efficiency:* It should not be possible to increase the allocation of a user without decreasing the allocation of at least another user. This property is important as it leads to maximizing system utilization subject to satisfying the other properties.

We briefly comment on the strategy-proofness and sharing incentive properties, which we believe are of special importance in datacenter environments. Anecdotal evidence from cloud operators that we have talked with indicates that strategy-proofness is important, as it is common for users to attempt to manipulate schedulers. For example, one of Yahoo!'s Hadoop MapReduce datacenters has different numbers of slots for map and reduce tasks. A user discovered that the map slots were contended, and therefore launched all his jobs as long reduce phases, which would manually do the work that MapReduce does in its map phase. Another big search company provided dedicated machines for jobs only if the users could guarantee high utilization. The company soon found that users would sprinkle their code with infinite loops to artificially inflate utilization levels.

Furthermore, any policy that satisfies the sharing incentive property also provides performance isolation, as it guarantees a minimum allocation to each user (*i.e.,* a user cannot do worse than owning $\frac{1}{n}$ of the cluster) irrespective of the demands of the other users.

It can be easily shown that in the case of a single resource, max-min fairness satisfies all the above properties. However, achieving these properties in the case of multiple resources and heterogeneous user demands is not trivial. For example, the preferred fair division mechanism in microeconomic theory, Competitive Equilibrium from Equal Incomes [22, 30, 33], is not strategy-proof (see Section 6.1.2).

In addition to the above properties, we consider four other nice-to-have properties:

- *Single resource fairness:* For a single resource, the solution should reduce to max-min fairness.

- *Bottleneck fairness:* If there is one resource that is percent-wise demanded most of by *every* user, then the solution should reduce to max-min fairness for that resource.

- *Population monotonicity:* When a user leaves the system and relinquishes her resources, none of the allocations of the remaining users should decrease.

- *Resource monotonicity:* If more resources are added to the system, none of the allocations of the existing users should decrease.

4 Dominant Resource Fairness (DRF)

We propose Dominant Resource Fairness (DRF), a new allocation policy for multiple resources that meets all four of the required properties in the previous section. For every user, DRF computes the share of each resource allocated to that user. The maximum among all shares of a user is called that user's *dominant share*, and the resource corresponding to the dominant share is called the *dominant resource*. Different users may have different dominant resources. For example, the dominant resource of a user running a computation-bound job is

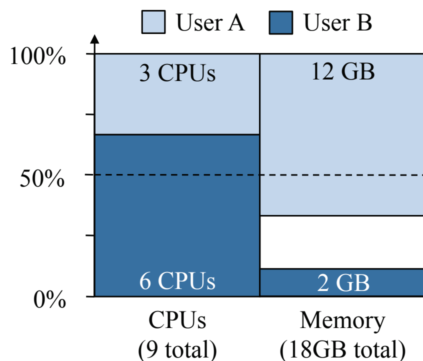


Figure 3: DRF allocation for the example in Section 4.1.

CPU, while the dominant resource of a user running an I/O-bound job is bandwidth.[1] DRF simply applies max-min fairness across users' dominant shares. That is, DRF seeks to maximize the smallest dominant share in the system, then the second-smallest, and so on.

We start by illustrating DRF with an example (§4.1), then present an algorithm for DRF (§4.2) and a definition of weighted DRF (§4.3). In Section 5, we present two other allocation policies: asset fairness, a straightforward policy that aims to equalize the aggregate resources allocated to each user, and competitive equilibrium from equal incomes (CEEI), a popular fair allocation policy preferred in the micro-economic domain [22, 30, 33].

In this section, we consider a computation model with n users and m resources. Each user runs individual tasks, and each task is characterized by a *demand vector*, which specifies the amount of resources required by the task, *e.g.,* $\langle$1 CPU, 4 GB$\rangle$. In general, tasks (even the ones belonging to the same user) may have different demands.

4.1 An Example

Consider a system with of 9 CPUs, 18 GB RAM, and two users, where user A runs tasks with demand vector $\langle$1 CPU, 4 GB$\rangle$, and user B runs tasks with demand vector $\langle$3 CPUs, 1 GB$\rangle$ each.

In the above scenario, each task from user A consumes $1/9$ of the total CPUs and $2/9$ of the total memory, so user A's dominant resource is memory. Each task from user B consumes $1/3$ of the total CPUs and $1/18$ of the total memory, so user B's dominant resource is CPU. DRF will equalize users' dominant shares, giving the allocation in Figure 3: three tasks for user A, with a total of $\langle$3 CPUs, 12 GB$\rangle$, and two tasks for user B, with a total of $\langle$6 CPUs, 2 GB$\rangle$. With this allocation, each user ends up with the *same* dominant share, *i.e.,* user A gets $2/3$ of RAM, while user B gets $2/3$ of the CPUs.

This allocation can be computed mathematically as follows. Let x and y be the number of tasks allocated

[1] A user may have the same share on multiple resources, and might therefore have multiple dominant resources.

Algorithm 1 DRF pseudo-code

```
R = ⟨r_1, ..., r_m⟩                          ▷ total resource capacities
C = ⟨c_1, ..., c_m⟩               ▷ consumed resources, initially 0
s_i (i = 1..n)                ▷ user i's dominant shares, initially 0
U_i = ⟨u_{i,1}, ..., u_{i,m}⟩ (i = 1..n) ▷ resources given to
                                                user i, initially 0

pick user i with lowest dominant share s_i
D_i ← demand of user i's next task
if C + D_i ≤ R then
    C = C + D_i                          ▷ update consumed vector
    U_i = U_i + D_i                   ▷ update i's allocation vector
    s_i = max_{j=1}^{m} {u_{i,j}/r_j}
else
    return                                      ▷ the cluster is full
end if
```

by DRF to users A and B, respectively. Then user A receives $\langle x$ CPU, $4x$ GB$\rangle$, while user B gets $\langle 3y$ CPU, y GB$\rangle$. The total amount of resources allocated to both users is $(x + 3y)$ CPUs and $(4x + y)$ GB. Also, the dominant shares of users A and B are $4x/18 = 2x/9$ and $3y/9 = y/3$, respectively (their corresponding shares of memory and CPU). The DRF allocation is then given by the solution to the following optimization problem:

$$\begin{array}{rcll} \max\,(x, y) & & & \text{(Maximize allocations)} \\ \text{subject to} & & & \\ x + 3y & \leq & 9 & \text{(CPU constraint)} \\ 4x + y & \leq & 18 & \text{(Memory constraint)} \\ \dfrac{2x}{9} & = & \dfrac{y}{3} & \text{(Equalize dominant shares)} \end{array}$$

Solving this problem yields[2] $x = 3$ and $y = 2$. Thus, user A gets $\langle$3 CPU, 12 GB$\rangle$ and B gets $\langle$6 CPU, 2 GB$\rangle$.

Note that DRF need not always equalize users' dominant shares. When a user's total demand is met, that user will not need more tasks, so the excess resources will be split among the other users, much like in max-min fairness. In addition, if a resource gets exhausted, users that do not need that resource can still continue receiving higher shares of the other resources. We present an algorithm for DRF allocation in the next section.

4.2 DRF Scheduling Algorithm

Algorithm 1 shows pseudo-code for DRF scheduling. The algorithm tracks the total resources allocated to each user as well as the user's dominant share, s_i. At each step, DRF picks the user with the lowest dominant share among those with tasks ready to run. If that user's task demand can be satisfied, *i.e.,* there are enough resources

[2] Note that given last constraint (*i.e.,* $2x/9 = y/3$) allocations x and y are simultaneously maximized.

Schedule	User A		User B		CPU	RAM
	res. shares	dom. share	res. shares	dom. share	total alloc.	total alloc.
User B	$\langle 0,\ 0\rangle$	**0**	$\langle 3/9,\ 1/18\rangle$	1/3	3/9	1/18
User A	$\langle 1/9,\ 4/18\rangle$	**2/9**	$\langle 3/9,\ 1/18\rangle$	1/3	4/9	5/18
User A	$\langle 2/9,\ 8/18\rangle$	4/9	$\langle 3/9,\ 1/18\rangle$	**1/3**	5/9	9/18
User B	$\langle 2/9,\ 8/18\rangle$	**4/9**	$\langle 6/9,\ 2/18\rangle$	2/3	8/9	10/18
User A	$\langle 3/9,\ 12/18\rangle$	**2/3**	$\langle 6/9,\ 2/18\rangle$	**2/3**	1	14/18

Table 1: Example of DRF allocating resources in a system with 9 CPUs and 18 GB RAM to two users running tasks that require ⟨1 CPU, 4 GB⟩ and ⟨3 CPUs, 1 GB⟩, respectively. Each row corresponds to DRF making a scheduling decision. A row shows the shares of each user for each resource, the user's dominant share, and the fraction of each resource allocated so far. DRF repeatedly selects the user with the lowest dominant share (indicated in bold) to launch a task, until no more tasks can be allocated.

available in the system, one of her tasks is launched. We consider the general case in which a user can have tasks with *different* demand vectors, and we use variable D_i to denote the demand vector of the next task user i wants to launch. For simplicity, the pseudo-code does not capture the event of a task finishing. In this case, the user releases the task's resources and DRF again selects the user with the smallest dominant share to run her task.

Consider the two-user example in Section 4.1. Table 1 illustrates the DRF allocation process for this example. DRF first picks B to run a task. As a result, the shares of B become $\langle 3/9,\ 1/18\rangle$, and the dominant share becomes $\max(3/9, 1/18) = 1/3$. Next, DRF picks A, as her dominant share is 0. The process continues until it is no longer possible to run new tasks. In this case, this happens as soon as CPU has been saturated.

At the end of the above allocation, user A gets ⟨3 CPU, 12 GB⟩, while user B gets ⟨6 CPU, 2 GB⟩, *i.e.,* each user gets $2/3$ of its dominant resource.

Note that in this example the allocation stops as soon as any resource is saturated. However, in the general case, it may be possible to continue to allocate tasks even after some resource has been saturated, as some tasks might not have any demand on the saturated resource.

The above algorithm can be implemented using a binary heap that stores each user's dominant share. Each scheduling decision then takes $O(\log n)$ time for n users.

4.3 Weighted DRF

In practice, there are many cases in which allocating resources equally across users is not the desirable policy. Instead, we may want to allocate more resources to users running more important jobs, or to users that have contributed more resources to the cluster. To achieve this goal, we propose Weighted DRF, a generalization of both DRF and weighted max-min fairness.

With Weighted DRF, each user i is associated a weight vector $W_i = \langle w_{i,1}, \dots, w_{i,m}\rangle$, where $w_{i,j}$ represents the weight of user i for resource j. The definition of a dominant share for user i changes to $s_i = \max_j\{u_{i,j}/w_{i,j}\}$, where $u_{i,j}$ is user i's share of resource j. A particular case of interest is when all the weights of user i are equal, *i.e.,* $w_{i,j} = w_i$, $(1 \leq j \leq m)$. In this case, the ratio between the dominant shares of users i and j will be simply w_i/w_j. If the weights of all users are set to 1, Weighted DRF reduces trivially to DRF.

5 Alternative Fair Allocation Policies

Defining a fair allocation in a multi-resource system is not an easy question, as the notion of "fairness" is itself open to discussion. In our efforts, we considered numerous allocation policies before settling on DRF as the only one that satisfies all four of the required properties in Section 3: sharing incentive, strategy-proofness, Pareto efficiency, and envy-freeness. In this section, we consider two of the alternatives we have investigated: Asset Fairness, a simple and intuitive policy that aims to equalize the aggregate resources allocated to each user, and Competitive Equilibrium from Equal Incomes (CEEI), the policy of choice for fairly allocating resources in the microeconomic domain [22, 30, 33]. We compare these policies with DRF in Section 5.3.

5.1 Asset Fairness

The idea behind Asset Fairness is that equal shares of different resources are worth the same, *i.e.,* that 1% of all CPUs worth is the same as 1% of memory and 1% of I/O bandwidth. Asset Fairness then tries to equalize the aggregate resource value allocated to each user. In particular, Asset Fairness computes for each user i the aggregate share $x_i = \sum_j s_{i,j}$, where $s_{i,j}$ is the share of resource j given to user i. It then applies max-min across users' aggregate shares, *i.e.,* it repeatedly launches tasks for the user with the minimum aggregate share.

Consider the example in Section 4.1. Since there are twice as many GB of RAM as CPUs (*i.e.,* 9 CPUs and 18 GB RAM), one CPU is worth twice as much as one GB of RAM. Supposing that one GB is worth \$1 and one CPU is worth \$2, it follows that user A spends \$6 for each task, while user B spends \$7. Let x and y be the number of tasks allocated by Asset Fairness to users A and B, respectively. Then the asset-fair allocation is

given by the solution to the following optimization problem:

$$\begin{aligned} \max\,(x, y) & \qquad \text{(Maximize allocations)} \\ \text{subject to} & \\ x + 3y &\leq 9 \;\text{(CPU constraint)} \\ 4x + y &\leq 18 \;\text{(Memory constraint)} \\ 6x &= 7y \;\text{(Every user spends the same)} \end{aligned}$$

Solving the above problem yields $x = 2.52$ and $y = 2.16$. Thus, user A gets $\langle$2.5 CPUs, 10.1 GB$\rangle$, while user B gets $\langle$6.5 CPUs, 2.2 GB$\rangle$, respectively.

While this allocation policy seems compelling in its simplicity, it has a significant drawback: it *violates* the sharing incentive property. As we show in Section 6.1.1, asset fairness can result in one user getting less than $1/n$ of all resources, where n is the total number of users.

5.2 Competitive Equilibrium from Equal Incomes

In microeconomic theory, the preferred method to fairly divide resources is Competitive Equilibrium from Equal Incomes (CEEI) [22, 30, 33]. With CEEI, each user receives initially $\frac{1}{n}$ of every resource, and subsequently, each user trades her resources with other users in a perfectly competitive market.[3] The outcome of CEEI is both envy-free and Pareto efficient [30].

More precisely, the CEEI allocation is given by the *Nash bargaining solution*[4] [22, 23]. The Nash bargaining solution picks the feasible allocation that maximizes $\prod_i u_i(a_i)$, where $u_i(a_i)$ is the utility that user i gets from her allocation a_i. To simplify the comparison, we assume that the utility that a user gets from her allocation is simply her dominant share, s_i.

Consider again the two-user example in Section 4.1. Recall that the dominant share of user A is $4x/18 = 2x/9$ while the dominant share of user B is $3y/9 = y/3$, where x is the number of tasks given to A and y is the number of tasks given to B. Maximizing the product of the dominant shares is equivalent to maximizing the product $x \cdot y$. Thus, CEEI aims to solve the following optimization problem:

$$\begin{aligned} \max\,(x \cdot y) & \qquad \text{(maximize Nash product)} \\ \text{subject to} & \\ x + 3y &\leq 9 \;\text{(CPU constraint)} \\ 4x + y &\leq 18 \;\text{(Memory constraint)} \end{aligned}$$

Solving the above problem yields $x = 45/11$ and $y = 18/11$. Thus, user A gets $\langle$4.1 CPUs, 16.4 GB$\rangle$, while user B gets $\langle$4.9 CPUs, 1.6 GB$\rangle$.

[3] A perfect market satisfies the price-taking (*i.e.,* no single user affects prices) and market-clearance (*i.e.,* matching supply and demand via price adjustment) assumptions.

[4] For this to hold, utilities have to be homogeneous, *i.e.,* $u(\alpha\, x) = \alpha\, u(x)$ for $\alpha > 0$, which is true in our case.

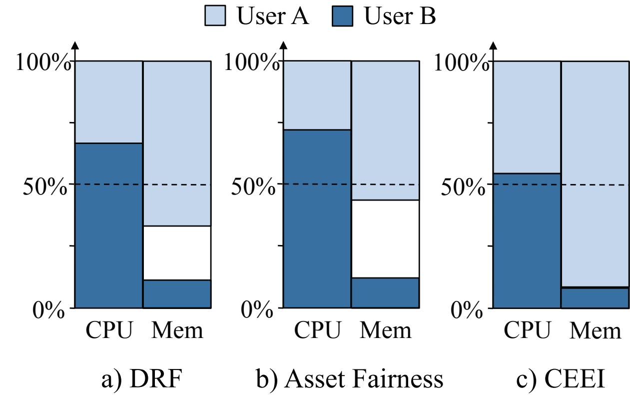


Figure 4: Allocations given by DRF, Asset Fairness and CEEI in the example scenario in Section 4.1.

Unfortunately, while CEEI is envy-free and Pareto efficient, it turns out that it is *not* strategy-proof, as we will show in Section 6.1.2. Thus, users can increase their allocations by lying about their resource demands.

5.3 Comparison with DRF

To give the reader an intuitive understanding of Asset Fairness and CEEI, we compare their allocations for the example in Section 4.1 to that of DRF in Figure 4.

We see that DRF equalizes the dominant shares of the users, *i.e.,* user A's memory share and user B's CPU share. In contrast, Asset Fairness equalizes the *total fraction of resources* allocated to each user, *i.e.,* the areas of the rectangles for each user in the figure. Finally, because CEEI assumes a perfectly competitive market, it finds a solution satisfying market clearance, where every resource has been allocated. Unfortunately, this exact property makes it possible to cheat CEEI: a user can claim she needs more of some underutilized resource even when she does not, leading CEEI to give more tasks overall to this user to achieve market clearance.

6 Analysis

In this section, we discuss which of the properties presented in Section 3 are satisfied by Asset Fairness, CEEI, and DRF. We also evaluate the accuracy of DRF when task sizes do not match the available resources exactly.

6.1 Fairness Properties

Table 2 summarizes the fairness properties that are satisfied by Asset Fairness, CEEI, and DRF. The Appendix contains the proofs of the main properties of DRF, while our technical report [14] contains a more complete list of results for DRF and CEEI. In the remainder of this section, we discuss some of the interesting missing entries in the table, *i.e.,* properties violated by each of these disciplines. In particular, we show through examples why Asset Fairness and CEEI lack the properties that they

Property	Allocation Policy Asset	CEEI	DRF
Sharing Incentive		✓	✓
Strategy-proofness	✓		✓
Envy-freeness	✓	✓	✓
Pareto efficiency	✓	✓	✓
Single Resource Fairness	✓	✓	✓
Bottleneck Fairness		✓	✓
Population Monotonicity	✓		✓
Resource Monotonicity			

Table 2: Properties of Asset Fairness, CEEI and DRF.

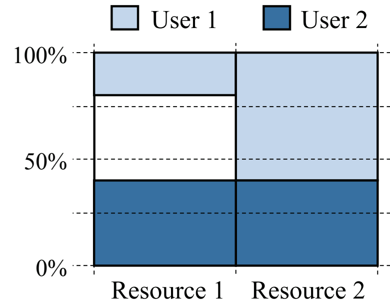


Figure 5: Example showing that Asset Fairness can fail to meet the sharing incentive property. Asset Fairness gives user 2 less than half of both resources.

do, and we prove that no policy can provide resource monotonicity without violating either sharing incentive or Pareto efficiency to explain why DRF lacks resource monotonicity.

6.1.1 Properties Violated by Asset Fairness

While being the simplest policy, Asset Fairness violates several important properties: sharing incentive, bottleneck fairness, and resource monotonicity. Next, we use examples to show the violation of these properties.

Theorem 1 *Asset Fairness violates the sharing incentive property.*

Proof Consider the following example, illustrated in Figure 5: two users in a system with $\langle 30, 30\rangle$ total resources have demand vectors $D_1 = \langle 1, 3\rangle$, and $D_2 = \langle 1, 1\rangle$. Asset fairness will allocate the first user 6 tasks and the second user 12 tasks. The first user will receive $\langle 6, 18\rangle$ resources, while the second will use $\langle 12, 12\rangle$. While each user gets an equal aggregate share of $\frac{24}{60}$, the second user gets less than half (15) of both resources. This violates the sharing incentive property, as the second user would be better off to statically partition the cluster and own half of the nodes. □

Theorem 2 *Asset Fairness violates the bottleneck fairness property.*

Proof Consider a scenario with a total resource vector of $\langle 21, 21\rangle$ and two users with demand vectors $D_1 = \langle 3, 2\rangle$ and $D_2 = \langle 4, 1\rangle$, making resource 1 the bottleneck resource. Asset fairness will give each user 3 tasks, equalizing their aggregate usage to 15. However, this only gives the first user $\frac{3}{7}$ of resource 1 (the contended bottleneck resource), violating bottleneck fairness. □

Theorem 3 *Asset fairness does not satisfy resource monotonicity.*

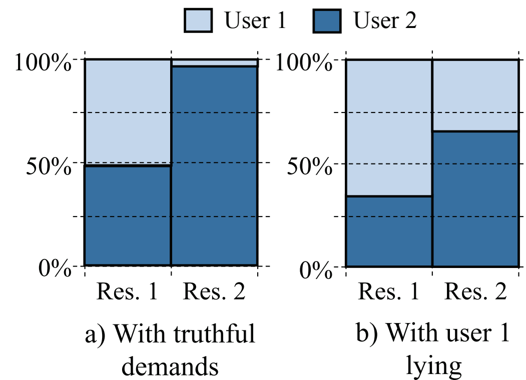


Figure 6: Example showing how CEEI violates strategy proofness. User 1 can increase her share by claiming that she needs more of resource 2 than she actually does.

Proof Consider two users A and B with demands $\langle 4, 2\rangle$ and $\langle 1, 1\rangle$ and 77 units of two resources. Asset fairness allocates A a total of $\langle 44, 22\rangle$ and B $\langle 33, 33\rangle$ equalizing their sum of shares to $\frac{66}{77}$. If resource two is doubled, both users' share of the second resource is halved, while the first resource is saturated. Asset fairness now decreases A's allocation to $\langle 42, 21\rangle$ and increases B's to $\langle 35, 35\rangle$, equalizing their shares to $\frac{42}{77} + \frac{21}{154} = \frac{35}{77} + \frac{35}{154} = \frac{105}{154}$. Thus resource monotonicity is violated. □

6.1.2 Properties Violated by CEEI

While CEEI is envy-free and Pareto efficient, it turns out that it is *not* strategy proof. Intuitively, this is because CEEI assumes a perfectly competitive market that achieves *market clearance*, *i.e.,* matching of supply and demand and allocation of all the available resources. This can lead to CEEI giving much higher shares to users that use more of a less-contended resource in order to fully utilize that resource. Thus, a user can claim that she needs more of some underutilized resource to increase her overall share of resources. We illustrate this below.

Theorem 4 *CEEI is not strategy-proof.*

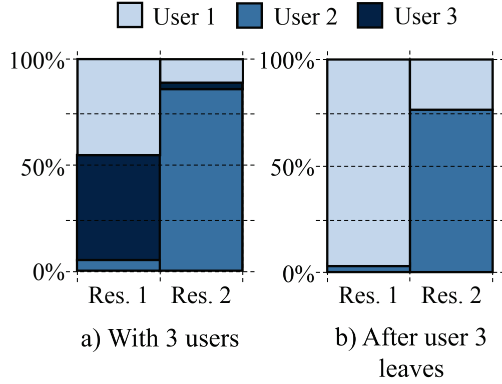


Figure 7: Example showing that CEEI violates population monotonicity. When user 3 leaves, CEEI changes the allocation from a) to b), lowering the share of user 2.

Proof Consider the following example, shown in Figure 6. Assume a total resource vector of $\langle 100, 100 \rangle$, and two users with demands $\langle 16, 1 \rangle$ and $\langle 1, 2 \rangle$. In this case, CEEI allocates $\frac{100}{31}$ and $\frac{1500}{31}$ tasks to each user respectively (approximately 3.2 and 48.8 tasks). If user 1 changes her demand vector to $\langle 16, 8 \rangle$, asking for more of resource 2 than she actually needs, CEEI gives the the users $\frac{25}{6}$ and $\frac{100}{3}$ tasks respectively (approximately 4.2 and 33.3 tasks). Thus, user 1 improves her number of tasks from 3.2 to 4.2 by lying about her demand vector. User 2 suffers because of this, as her task allocation decreases. □

In addition, for the same intuitive reason (market clearance), we have the following result:

Theorem 5 *CEEI violates population monotonicity.*

Proof Consider the total resource vector $\langle 100, 100 \rangle$ and three users with the following demand vectors $D_1 = \langle 4, 1 \rangle$, $D_2 = \langle 1, 16 \rangle$, and $D_3 = \langle 16, 1 \rangle$ (see Figure 7). CEEI will yield the allocation $A_1 = \langle 11.3, 5.4, 3.1 \rangle$, where the numbers in parenthesis represent the number of tasks allocated to each user. If user 3 leaves the system and relinquishes her resource, CEEI gives the new allocation $A_2 = \langle 23.8, 4.8 \rangle$, which has made user 2 worse off than in A_1. □

6.1.3 Resource Monotonicity vs. Sharing Incentives and Pareto efficiency

As shown in Table 2, DRF achieves all the properties except resource monotonicity. Rather than being a limitation of DRF, this is a consequence of the fact that sharing incentive, Pareto efficiency, and resource monotonicity cannot be achieved simultaneously. Since we consider the first two of these properties to be more important (see Section 3) and since adding new resources to a system is a relatively rare event, we chose to satisfy sharing incentive and Pareto efficiency, and give up resource monotonicity. In particular, we have the following result.

Theorem 6 *No allocation policy that satisfies the sharing incentive and Pareto efficiency properties can also satisfy resource monotonicity.*

Proof We use a simple example to prove this property. Consider two users A and B with symmetric demands $\langle 2, 1 \rangle$, and $\langle 1, 2 \rangle$, respectively, and assume equal amounts of both resources. Sharing incentive requires that user A gets at least half of resource 1 and user B gets half of resource 2. By Pareto efficiency, we know that at least one of the two users must be allocated more resources. Without loss of generality, assume that user A is given more than half of resource 1 (a symmetric argument holds if user B is given more than half of resource 2). If the total amount of resource 2 is now increased by a factor of 4, user B is no longer getting its guaranteed share of half of resource 2. Now, the only feasible allocation that satisfies the sharing incentive is to give both users half of resource 1, which would require decreasing user 1's share of resource 1, thus violating resource monotonicity. □

This theorem explains why both DRF and CEEI violate resource monotonicity.

6.2 Discrete Resource Allocation

So far, we have implicitly assumed one big resource pool whose resources can be allocated in arbitrarily small amounts. Of course, this is often not the case in practice. For example, clusters consist of many small machines, where resources are allocated to tasks in discrete amounts. In the reminder of this section, we refer to these two scenarios as the *continuous*, and the *discrete* scenario, respectively. We now turn our attention to how fairness is affected in the discrete scenario.

Assume a cluster consisting of K machines. Let *max-task* denote the maximum demand vector across all demand vectors, *i.e., max-task* $= \langle \max_i\{d_{i,1}\}, \max_i\{d_{i,2}\}, \cdots, \max_i\{d_{i,m}\} \rangle$. Assume further that any task can be scheduled on every machine, *i.e.,* the total amount of resources on each machine is at least max-task. We only consider the case when each user has strictly positive demands. Given these assumptions, we have the following result.

Theorem 7 *In the discrete scenario, it is possible to allocate resources such that the difference between the allocations of any two users is bounded by one max-task compared to the continuous allocation scenario.*

Proof Assume we start allocating resources on one machine at a time, and that we always allocate a task to the user with the lowest dominant share. As long as there is at least a *max-task* available on the first machine, we

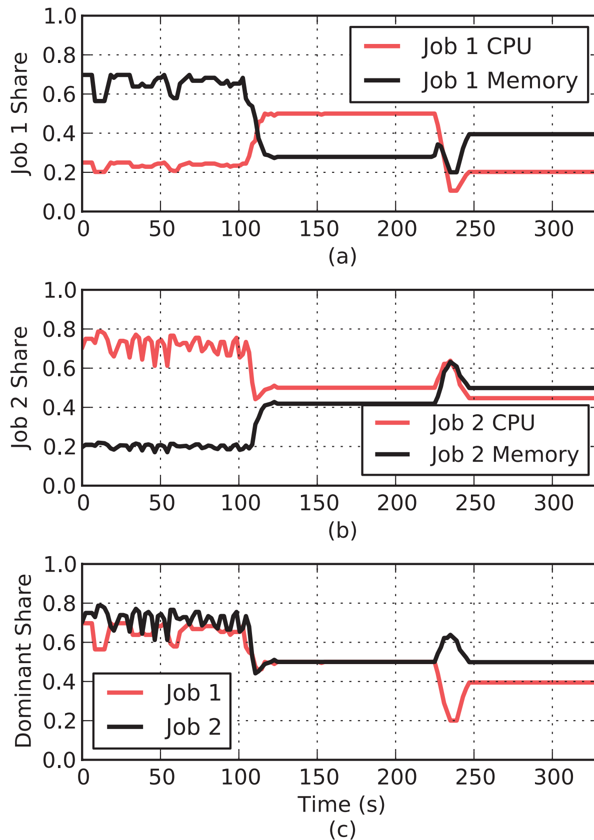


Figure 8: CPU, memory and dominant share for two jobs.

continue to allocate a task to the next user with least dominant share. Once the available resources on the first machine become less than a *max-task* size, we move to the next machine and repeat the process. When the allocation completes, the difference between two user's allocations of their dominant resources compared to the continuous scenario is at most *max-task*. If this were not the case, then some user A would have more than *max-task* discrepancy w.r.t. to another user B. However, this cannot be the case, because the last time A was allocated a task, B should have been allocated a task instead. □

7 Experimental Results

This section evaluates DRF through micro- and macro-benchmarks. The former is done through experiments running an implementation of DRF in the Mesos cluster resource manager [16]. The latter is done using trace-driven simulations.

We start by showing how DRF dynamically adjusts the shares of jobs with different resource demands in Section 7.1. In Section 7.2, we compare DRF against slot-level fair sharing (as implemented by Hadoop Fair Scheduler [34] and Quincy [18]), and CPU-only fair sharing. Finally, in Section 7.3, we use Facebook traces to compare DRF and the Hadoop's Fair Scheduler in terms of utilization and job completion time.

7.1 Dynamic Resource Sharing

In our first experiment, we show how DRF dynamically shares resources between jobs with different demands. We ran two jobs on a 48-node Mesos cluster on Amazon EC2, using "extra large" instances with 4 CPU cores and 15 GB of RAM. We configured Mesos to allocate up to 4 CPUs and 14 GB of RAM on each node, leaving 1 GB for the OS. We submitted two jobs that launched tasks with different resource demands at different times during a 6-minute interval.

Figures 8 (a) and 8 (b) show the CPU and memory allocations given to each job as a function of time, while Figure 8 (c) shows their dominant shares. In the first 2 minutes, job 1 uses $\langle$1 CPU, 10 GB RAM$\rangle$ per task and job 2 uses $\langle$1 CPU, 1 GB RAM$\rangle$ per task. Job 1's dominant resource is RAM, while job 2's dominant resource is CPU. Note that DRF equalizes the jobs' shares of their dominant resources. In addition, because jobs have different dominant resources, their dominant shares exceed 50%, *i.e.,* job 1 uses around 70% of the RAM while job 2 uses around 75% of the CPUs. Thus, the jobs benefit from running in a shared cluster as opposed to taking half the nodes each. This captures the essence of the sharing incentive property.

After 2 minutes, the task sizes of both jobs change, to $\langle$2 CPUs, 4 GB$\rangle$ for job 1 and $\langle$1 CPU, 3 GB$\rangle$ for job 2. Now, both jobs' dominant resource is CPU, so DRF equalizes their CPU shares. Note that DRF switches allocations dynamically by having Mesos offer resources to the job with the smallest dominant share as tasks finish.

Finally, after 2 more minutes, the task sizes of both jobs change again: $\langle$1 CPU, 7 GB$\rangle$ for job 1 and $\langle$1 CPU, 4 GB$\rangle$ for job 2. Both jobs' dominant resource is now memory, so DRF tries to equalize their memory shares. The reason the shares are not exactly equal is due to resource fragmentation (see Section 6.2).

7.2 DRF vs. Alternative Allocation Policies

We next evaluate DRF with respect to two alternative schemes: slot-based fair scheduling (a common policy in current systems, such as the Hadoop Fair Scheduler [34] and Quincy [18]) and (max-min) fair sharing applied only to a single resource (CPU). For the experiment, we ran a 48-node Mesos cluster on EC2 instances with 8 CPU cores and 7 GB RAM each. We configured Mesos to allocate 8 CPUs and 6 GB RAM on each node, leaving 1 GB free for the OS. We implemented these three scheduling policies as Mesos allocation modules.

We ran a workload with two classes of users, representing two organizational entities with different workloads. One of the entities had four users submitting small jobs with task demands $\langle$1 CPU, 0.5 GB$\rangle$. The other en-

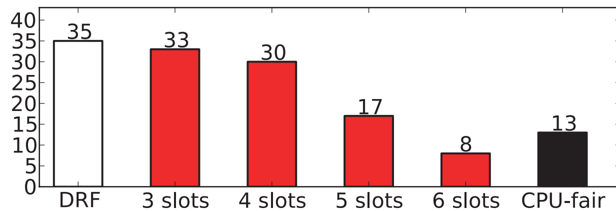


Figure 9: Number of large jobs completed for each allocation scheme in our comparison of DRF against slot-based fair sharing and CPU-only fair sharing.

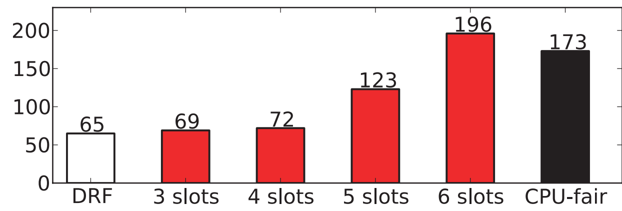


Figure 11: Average response time (in seconds) of large jobs for each allocation scheme in our comparison of DRF against slot-based fair sharing and CPU-only fair sharing.

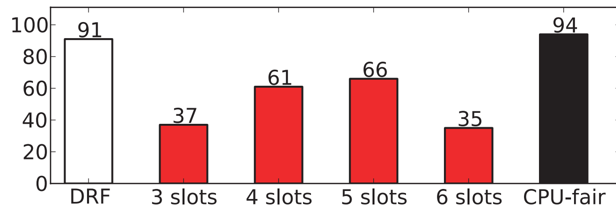


Figure 10: Number of small jobs completed for each allocation scheme in our comparison of DRF against slot-based fair sharing and CPU-only fair sharing.

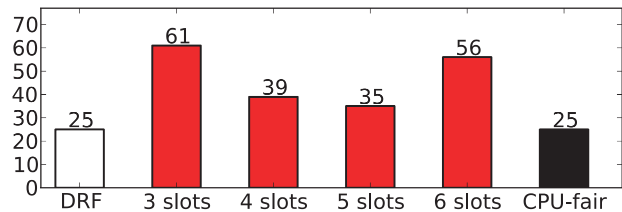


Figure 12: Average response time (in seconds) of small jobs for each allocation scheme in our comparison of DRF against slot-based fair sharing and CPU-only fair sharing.

tity had four users submitting large jobs with task demands $\langle$2 CPUs, 2 GB$\rangle$. Each job consisted of 80 tasks. As soon as a job finished, the user would launch another job with similar demands. Each experiment ran for ten minutes. At the end, we computed the number of completed jobs of each type, as well as their response times.

For the slot-based allocation scheme, we varied the number of slots per machine from 3 to 6 to see how it affected performance. Figures 9 through 12 show our results. In Figures 9 and 10, we compare the number of jobs of each type completed for each scheduling scheme in ten minutes. In Figures 11 and 12, we compare average response times.

Several trends are apparent from the data. First, with slot-based scheduling, both the throughput and job response times are worse than with DRF, regardless of the number of slots. This is because with a low slot count, the scheduler can undersubscribe nodes (*e.g.,*, launch only 3 small tasks on a node), while with a large slot count, it can oversubscribe them (*e.g.,* launch 4 large tasks on a node and cause swapping because each task needs 2 GB and the node only has 6 GB). Second, with fair sharing at the level of CPUs, the number of small jobs executed is similar to DRF, but there are much fewer large jobs executed, because memory is overcommitted on some machines and leads to poor performance for all the high-memory tasks running there. Overall, the DRF-based scheduler that is aware of both resources has the lowest response times and highest overall throughput.

7.3 Simulations using Facebook Traces

Next we use log traces from a 2000-node cluster at Facebook, containing data for a one week period (October 2010). The data consists of Hadoop MapReduce jobs. We assume task duration, CPU usage, and memory consumption is identical as in the original trace. The traces are simulated on a smaller cluster of 400 nodes to reach higher utilization levels, such that fairness becomes relevant. Each node in the cluster consists of 12 slots, 16 cores, and 32 GB memory. Figure 13 shows a short 300 second sub-sample to visualize how CPU and memory utilization looks for the same workload when using DRF compared to Hadoop's fair scheduler (slot). As shown in the figure, DRF provides higher utilization, as it is able to better match resource allocations with task demands.

Figure 14 shows the reduction of the average job completion times for DRF as compared to the Hadoop fair scheduler. The workload is quite heavy on small jobs, which experience no improvements (*i.e.,* -3%). This is because small jobs typically consist of a single execution phase, and the completion time is dominated by the longest task. Thus completion time is hard to improve for such small jobs. In contrast, the completion times of the larger jobs reduce by as much as 66%. This is because these jobs consists of many phases, and thus they can benefit from the higher utilization achieved by DRF.

8 Related Work

We briefly review related work in computer science and economics.

While many papers in computer science focus on multi-resource fairness, they are only considering multiple instances of the *same* interchangeable resource, *e.g.,* CPU [6, 7, 35], and bandwidth [10, 20, 21]. Unlike these approaches, we focus on the allocation of resources of different types.

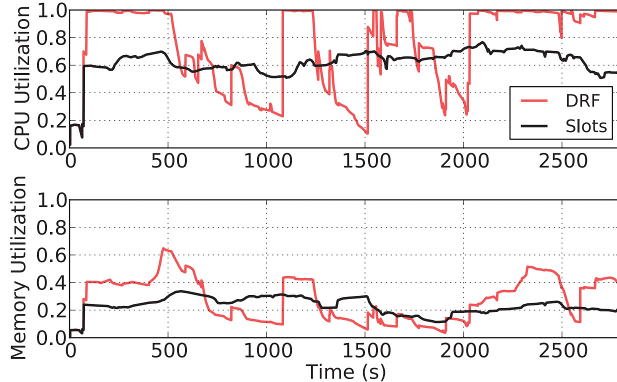


Figure 13: CPU and memory utilization for DRF and slot fairness for a trace from a Facebook Hadoop cluster.

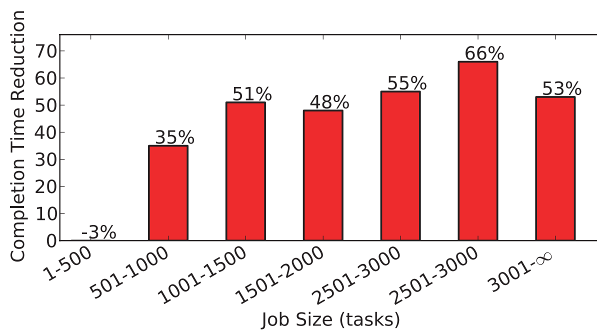


Figure 14: Average reduction of the completion times for different job sizes for a trace from a Facebook Hadoop cluster.

Quincy [18] is a scheduler developed in the context of the Dryad cluster computing framework [17]. Quincy achieves fairness by modeling the fair scheduling problem as a min-cost flow problem. Quincy does not currently support multi-resource fairness. In fact, as mentioned in the discussion section of the paper [18, pg. 17], it appears difficult to incorporate multi-resource requirements into the min-cost flow formulation.

Hadoop currently provides two fair sharing schedulers [1, 2, 34]. Both these schedulers allocate resources at the slot granularity, where a slot is a fixed fraction of the resources on a machine. As a result, these schedulers cannot always match the resource allocations with the tasks' demands, especially when these demands are widely heterogeneous. As we have shown in Section 7, this mismatch may lead to either low cluster utilization or poor performance due to resource oversubscription.

In the microeconomic literature, the problem of equity has been studied within and outside of the framework of game theory. The books by Young [33] and Moulin [22] are entirely dedicated to these topics and provide good introductions. The preferred method of fair division in microeconomics is CEEI [3, 33, 22], as introduced by Varian [30]. We have therefore devoted considerable attention to it in Section 5.2. CEEI's main drawback compared to DRF is that it is not strategy-proof. As a result, users can manipulate the scheduler by lying about their demands.

Many of the fair division policies proposed in the microeconomics literature are based on the notion of utility and, hence, focus on the single metric of utility. In the economics literature, max-min fairness is known as the lexicographic ordering [26, 25] (leximin) of utilities.

The question is what the user utilities are in the multi-resource setting, and how to compare such utilities. One natural way is to define utility as the number of tasks allocated to a user. But modeling utilities this way, together with leximin, violates many of the fairness properties we proposed. Viewed in this light, DRF makes two contributions. First, it suggests using the dominant share as a proxy for utility, which is equalized using the standard leximin ordering. Second, we prove that this scheme is strategy-proof for such utility functions. Note that the leximin ordering is a lexicographic version of the Kalai-Smorodinsky (KS) solution [19]. Thus, our result shows that KS is strategy-proof for such utilities.

9 Conclusion and Future Work

We have introduced Dominant Resource Fairness (DRF), a fair sharing model that generalizes max-min fairness to multiple resource types. DRF allows cluster schedulers to take into account the heterogeneous demands of datacenter applications, leading to both fairer allocation of resources and higher utilization than existing solutions that allocate identical resource slices (slots) to all tasks. DRF satisfies a number of desirable properties. In particular, DRF is strategy-proof, so that users are incentivized to report their demands accurately. DRF also incentivizes users to share resources by ensuring that users perform at least as well in a shared cluster as they would in smaller, separate clusters. Other schedulers that we investigated, as well as alternative notions of fairness from the microeconomic literature, fail to satisfy all of these properties.

We have evaluated DRF by implementing it in the Mesos resource manager, and shown that it can lead to better overall performance than the slot-based fair schedulers that are commonly in use today.

9.1 Future Work

There are several interesting directions for future research. First, in cluster environments with discrete tasks, one interesting problem is to minimize resource fragmentation without compromising fairness. This problem is similar to bin-packing, but where one must pack as many items (tasks) as possible subject to meeting DRF. A second direction involves defining fairness when tasks have placement constraints, such as machine preferences. Given the current trend of multi-core machines,

a third interesting research direction is to explore the use of DRF as an operating system scheduler. Finally, from a microeconomic perspective, a natural direction is to investigate whether DRF is the only possible strategy-proof policy for multi-resource fairness, given other desirable properties such Pareto efficiency.

10 Acknowledgements

We thank Eric J. Friedman, Hervé Moulin, John Wilkes, and the anonymous reviewers for their invaluable feedback. We thank Facebook for making available their traces. This research was supported by California MICRO, California Discovery, the Swedish Research Council, the Natural Sciences and Engineering Research Council of Canada, a National Science Foundation Graduate Research Fellowship,[5] and the RAD Lab sponsors: Google, Microsoft, Oracle, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

References

[1] Hadoop Capacity Scheduler. `http://hadoop.apache.org/common/docs/r0.20.2/capacity_scheduler.html`.

[2] Hadoop Fair Scheduler. `http://hadoop.apache.org/common/docs/r0.20.2/fair_scheduler.html`.

[3] Personal communication with Hervé Moulin.

[4] A. K. Agrawala and R. M. Bryant. Models of memory scheduling. In *SOSP '75*, 1975.

[5] J. Axboe. Linux Block IO – Present and Future (Completely Fair Queueing). In *Ottawa Linux Symposium 2004*, pages 51–61, 2004.

[6] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[7] S. K. Baruah, J. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *IPPS '95*, 1995.

[8] J. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *INFOCOM*, 1996.

[9] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, second edition, 1992.

[10] J. M. Blanquer and B. Özden. Fair queuing for aggregated multiple links. *SIGCOMM '01*, 31(4):189–197, 2001.

[11] B. Caprita, W. C. Chan, J. Nieh, C. Stein, and H. Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *USENIX Annual Technical Conference*, 2005.

[12] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *SIGCOMM '89*, pages 1–12, New York, NY, USA, 1989. ACM.

[13] D. Foley. Resource allocation and the public sector. *Yale Economic Essays*, 7(1):73–76, 1967.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. Technical Report UCB/EECS-2011-18, EECS Department, University of California, Berkeley, Mar 2011.

[15] P. Goyal, H. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Transactions on Networking*, 5(5):690–704, Oct. 1997.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[17] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys 07*, 2007.

[18] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *SOSP '09*, 2009.

[19] E. Kalai and M. Smorodinsky. Other Solutions to Nash's Bargaining Problem. *Econometrica*, 43(3):513–518, 1975.

[20] J. M. Kleinberg, Y. Rabani, and É. Tardos. Fairness in routing and load balancing. *J. Comput. Syst. Sci.*, 63(1):2–20, 2001.

[21] Y. Liu and E. W. Knightly. Opportunistic fair scheduling over multiple wireless channels. In *INFOCOM*, 2003.

[22] H. Moulin. *Fair Division and Collective Welfare*. The MIT Press, 2004.

[23] J. Nash. The Bargaining Problem. *Econometrica*, 18(2):155–162, April 1950.

[24] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control - the single node case. *ACM/IEEE Transactions on Networking*, 1(3):344–357, June 1993.

[25] E. A. Pazner and D. Schmeidler. Egalitarian equivalent allocations: A new concept of economic equity. *Quarterly Journal of Economics*, 92:671–687, 1978.

[26] A. Sen. Rawls Versus Bentham: An Axiomatic Examination of the Pure Distribution Problem. *Theory and Decision*, 4(1):301–309, 1974.

[27] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round robin. *IEEE Trans. Net*, 1996.

[28] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE RTSS 96*, 1996.

[29] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM*, 1998.

[30] H. Varian. Equity, envy, and efficiency. *Journal of Economic Theory*, 9(1):63–91, 1974.

[31] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional Share Resource Management*. PhD thesis, MIT, Laboratory of Computer Science, Sept. 1995. MIT/LCS/TR-667.

[32] C. A. Waldspurger and W. E. Weihl. Lottery scheduling:

[5] Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

flexible proportional-share resource management. In *OSDI '94*, 1994.

[33] H. P. Young. *Equity: in theory and practice*. Princeton University Press, 1994.

[34] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *EuroSys 10*, 2010.

[35] D. Zhu, D. Mossé, and R. G. Melhem. Multiple-Resource Periodic Scheduling Problem: how much fairness is necessary? In *IEEE RTSS*, 2003.

A Appendix: DRF Properties

In this appendix, we present the main properties of DRF. The technical report [14] contains a more complete list of results for DRF and CEEI. For context, the following table summarizes the properties satisfied by Asset Fairness, CEEI, and DRF, respectively.

In this section, we assume that all users have an unbounded number of tasks. In addition, we assume that all tasks of a user have the same demand vector, and we will refer to this vector as the user's demand vector.

Next, we present *progressive filling* [9], a simple technique to achieve DRF allocation when all resources are arbitrary divisible. This technique is instrumental in proving our results.

A.1 Progressive Filling for DRF

Progressive filling is an idealized algorithm to achieve max-min fairness in a system in which resources can be allocated in arbitrary small amounts [9, pg 450]. It was originally used in a networking context, but we now adapt it to our problem domain. In the case of DRF, progressive filling increases all users' dominant shares at the same rate, while increasing their other resource allocations *proportionally* to their task demand vectors, until at least one resource is saturated. At this point, the allocations of all users using the saturated resource are frozen, and progressive filling continues recursively after eliminating these users. In this case, progressive filling terminates when there are no longer users whose dominant shares can be increased.

Progressive filling for DRF is equivalent to the scheduling algorithm presented in Figure 1 after appropriately scaling the users' demand vectors. In particular, each user's demand vector is scaled such that allocating resources to a user according to her scaled demand vector will increase her dominant share by a fixed ϵ, which is the same for all users. Let $D_i = \langle d_{i,1}, d_{i,2}, \ldots, d_{i,m} \rangle$ be the demand vector of user i, let r_k be her dominant share[6], and let $s_i = \frac{d_{i,k}}{r_k}$ be her dominant share. We then scale the demand vector of user i by $\frac{\epsilon}{s_i}$, *i.e.,* $D'_i = \frac{\epsilon}{s_i} D_i = \frac{\epsilon}{s_i} \langle d_{i,1}, d_{i,2}, \ldots, d_{i,m} \rangle$. Thus, every time a task of user i is selected, she is allocated an amount $\frac{\epsilon}{s_i} d_{i,k} = \epsilon \cdot r_k$ of the dominant resource. This means that the share of the dominant resource of user i increases by $(\epsilon \cdot r_k)/r_k = \epsilon$, as expected.

[6] Recall that in this section we assume that all tasks of a user have the same demand vector.

A.2 Allocation Properties

We start with a preliminary result.

Lemma 8 *Every user in a DRF allocation has at least one saturated resource.*

Proof Assume this is not the case, *i.e.,* none of the resources used by user i is saturated. However, this contradicts the assumption that progressive filling has completed the computation of the DRF allocation. Indeed, as long as none of the resources of user i are saturated, progressive filling will continue to increase the allocations of user i (and of all the other users sharing only non-saturated resources). □

Recall that progressive filling always allocates the resources to a user *proportionally* to the user's demand vector. More precisely, let $D_i = \langle d_{i,1}, d_{i,2}, \ldots, d_{i,m} \rangle$ be the demand vector of user i. Then, at any time t during the progressive filling process, the allocation of user i is proportional to the demand vector,

$$A_i(t) = \alpha_i(t) \cdot D_i = \alpha_i(t) \cdot \langle d_{i,1}, d_{i,2}, \ldots, d_{i,m} \rangle \quad (1)$$

where $\alpha_i(t)$ is a positive scalar.

Now, we are in position to prove the DRF properties.

Theorem 9 *DRF is Pareto efficient.*

Proof Assume user i can increase her dominant share, s_i, without decreasing the dominant share of anyone else. According to Lemma 8, user i has at least one saturated resource. If no other user is using the saturated resource, then we are done as it would be impossible to increase i's share of the saturated resource. If other users are using the saturated resource, then increasing the allocation of i would result in decreasing the allocation of at least another user j sharing the same saturated resource. Since under progressive filling, the resources allocated by any user are proportional to her demand vector (see Eq. 1), decreasing the allocation of any resource used by user i will also decrease i's dominant share. This contradicts our hypothesis, and therefore proves the result. □

Theorem 10 *DRF satisfies the sharing incentive and bottleneck fairness properties.*

Proof Consider a system consisting of n users. Assume resource k is the first one being saturated by using progressive filling. Let i be the user allocating the largest share on resource k, and let $t_{i,k}$ denote her share of k. Since resource k is saturated, we have trivially $t_{i,k} \geq \frac{1}{n}$.

Furthermore, by the definition of the dominant share, we have $s_i \geq t_{i,k} \geq \frac{1}{n}$. Since progressive filling increases the allocation of each user's dominant resource at the *same* rate, it follows that each user gets at least $\frac{1}{n}$ of her dominant resource. Thus, DRF satisfies the sharing incentive property. If all users have the same dominant resource, each user gets exactly $\frac{1}{n}$ of that resource. As a result, DRF satisfies the bottleneck fairness property as well. □

Theorem 11 *Every DRF allocation is envy-free.*

Proof Assume by contradiction that user i envies another user j. For user i to envy another user j, user j must have a strictly higher share of every resource that i wants; otherwise i cannot run more tasks under j's allocation. This means that user j's dominant share is strictly larger than user i's dominant share. Since every resource allocated to user i is also allocated to user j, this means that user j cannot reach its saturated resource *after* user i, *i.e.,* $t_j \leq t_i$, where t_k is the time that user k's allocation gets frozen due to saturation. However, if $t_j \leq t_i$, under progressive filling, the dominant shares of users j and i will be equal at time t_j, after which the dominant share of user i can only increase, violating the hypothesis. □

Theorem 12 (Strategy-proofness) *A user cannot increase her dominant share in DRF by altering her true demand vector.*

Proof Assume user i can increase her dominant share by using a demand vector $\hat{d}_i \neq d_i$. Let $a_{i,j}$ and $\hat{a}_{i,j}$ denote the amount of resource j user i is allocated using progressive filling when the user uses the vector d_i and $\hat{d}_i$, respectively. For user i to be better off using $\hat{d}_i$, we need that $\hat{a}_{i,k} > a_{i,k}$ for every resource k where $d_{i,k} > 0$. Let r denote the first resource that becomes saturated for user i when she uses the demand vector d_i. If no other user is allocated resource r ($a_{j,r} = 0$ for all $j \neq i$), this contradicts the hypothesis as user i is already allocated the entire resource r, and thus cannot increase her allocation of r using another demand vector $\hat{d}_i$. Thus, assume there are other users that have been allocated r ($a_{j,r} > 0$ for some $j \neq i$). In this case, progressive filling will eventually saturate r at time t when using d_i, and at time t' when using demand $\hat{d}_i$. Recall that the dominant share is the maximum of a user's shares, thus i must have a higher dominant share in the allocation $\hat{a}$ than in a. Thus, $t' > t$, as progressive filling increases the dominant share at a constant rate. This implies that i—when using $\hat{d}$—does not saturate any resource before time t', and hence does not affect other user's allocation before time t'. Thus, when i uses $\hat{d}$, any user m using resource r has allocation $a_{m,r}$ at time t. Therefore, at time t, there is only $a_{i,r}$ amount of r left for user i, which contradicts the assumption that $\hat{a}_{i,r} > a_{i,r}$. □

The strategy-proofness of DRF shows that a user will not be better off by demanding resources that she does not need. The following example shows that excess demand can in fact hurt user's allocation, leading to a lower dominant share. Consider a cluster with two resources, and 10 users, the first with demand vector $\langle 1, 0 \rangle$ and the rest with demand vectors $\langle 0, 1 \rangle$. The first user gets the entire first resource, while the rest of the users each get $\frac{1}{9}$ of the second resource. If user 1 instead changes her demand vector to $\langle 1, 1 \rangle$, she can only be allocated $\frac{1}{10}$ of each resource and the rest of the users get $\frac{1}{10}$ of the second resource.

In practice, the situation can be exacerbated as resources in datacenters are typically partitioned across different physical machines, leading to fragmentation. Increasing one's demand artificially might lead to a situation in which, while there are enough resources on the whole, there are not enough on any single machine to satisfy the new demand. See Section 6.2 for more information.

Next, for simplicity we assume *strictly positive demand vectors*, *i.e.,* the demand of every user for every resource is non-zero.

Theorem 13 *Given strictly positive demand vectors, DRF guarantees that every user gets the same dominant share,* i.e., *every DRF allocation ensures $s_i = s_j$, for all users i and j.*

Proof Progressive filling will start increasing every users' dominant resource allocation at the same rate until one of the resources becomes saturated. At this point, no more resources can be allocated to any user as every user demands a positive amount of the saturated resource. □

Theorem 14 *Given strictly positive demands, DRF satisfies population monotonicity.*

Proof Consider any DRF allocation. Non-zero demands imply that all users have the same saturated resource(s). Consider removing a user and relinquishing her currently allocated resources, which is some amount of every resource. Since all users have the same dominant share α, any new allocation which decreases any user i's dominant share below α would, due to Pareto efficiency, have to allocate another user j a dominant share of more than α. The resulting allocation would violate max-min fairness, as it would be possible to increase i's dominant share by decreasing the allocation of j, who already has a higher dominant share than i. □

However, we note that in the absence of strictly positive demand vectors, DRF no longer satisfies the population monotonicity property [14].

PIE in the Sky: Online Passive Interference Estimation for Enterprise WLANs

Vivek Shrivastava
Nokia Research Center *

Shravan Rayanchu, Suman Banerjee
University of Wisconsin-Madison

Konstantina Papagiannaki
Intel Labs, Pittsburgh

Abstract

Trends in enterprise WLAN usage and deployment point to the need for tools that can capture interference in real time. A tool for interference estimation can not only enable WLAN managers to improve network performance by dynamically adjusting operating parameters like the channel of operation and transmit power of access points, but also diagnose and potentially proactively fix problems. In this paper, we present the design, implementation, and evaluation of a Passive Interference Estimator (PIE) that can dynamically generate fine-grained interference estimates across an entire WLAN. PIE introduces no measurement traffic, and yet provides an accurate estimate of WLAN interference tracking changes caused by client mobility, dynamic traffic loads, and varying channel conditions. Our experiments conducted on two different testbeds, using both controlled and real traffic patterns, show that PIE is not only able to provide high accuracy but also operate beyond the limitations of prior tools. It helps with performance diagnosis and real-time WLAN optimization, we describe its use in multiple WLAN optimization applications: channel assignment, transmit power control, and data scheduling.

1 Introduction

Radio interference remains a key performance bottleneck for enterprise WLANs [25]. In spite of significant progress in planning, deploying, and managing enterprise WLANs, administrators today have very tools that can help them understand how much interference exists in their network, and how interference patterns evolve over time. Building an on-line tool for enterprise-wide WLAN interference estimation is particularly challenging, because interference is highly dynamic in nature. Each time a new client arrives, departs, moves, or changes its traffic pattern, the number of other nodes in the network it interferes with (and the degree to which it interferes) changes. Further, wireless channel conditions are never static but continuously evolve with changes in the environment, e.g., even with the opening or closing of a door, people walking, etc.

The goal of this paper is to answer the following question: *Given an enterprise WLAN consisting of a number of Access Points (APs) and mobile clients, compute its real-time conflict graph, i.e., identify the precise set of nodes that interfere with each other and the degree to which they do so at any specified point of time.*

Applications of interference estimation: This problem of interference estimation is fundamental to understanding the behavior of any wireless network. Further, interference estimates and the conflict graph serve as important inputs to many WLAN configuration problems, e.g., channel assignment for each AP, transmit power selection, and even emerging strategies for data scheduling across the enterprise WLAN [22].

A number of research efforts have made significant progress toward this tool building goal. Prior techniques for interference estimation mainly employ active probing (interference maps [15] and micro-probing [3]) and suffer from three main problems: a) they incur moderate to significant measurement overhead and cannot be employed to continuously obtain interference information across time, b) they offer limited visibility into the root cause of interference, c) they often require specific client modifications. While some recent work has also explored the potential for passive interference estimation, it is mostly limited to offline trace collection and analysis, and thus cannot be employed in real time.

In this paper, we explore an alternate design for a practical online interference estimation mechanism, one that does not impose any active measurement traffic on the WLAN. It is completely passive in nature, and estimates interference by simply observing ongoing traffic at the different APs. Specifically, we present the design, implementation, and detailed evaluation of a Passive Interference Estimator (PIE) system.

Our work is inspired by two key passive WLAN monitoring approaches proposed earlier: Jigsaw [8, 9] and WIT [13]. These systems provide us with two useful building blocks: (i) a platform for capturing wireless traffic and merging traces collected from different vantage points and (ii) specific tools to infer interesting properties about the 802.11 network from such merged traffic traces. However, both these research efforts stop short of addressing our goal of designing a real-time interference estimation tool. The key features of PIE are:

1. It captures dynamic interference information quickly and robustly: PIE captures interference information across the entire WLAN within a few hundred milliseconds. It can effectively identify the real interfer-

*Vivek Shrivastava worked on this project as a PhD student at UW-Madison

ers when multiple overlapping transmitters are present.

2. It uses real traffic patterns: PIE is passive; it estimates interference using actual traffic patterns in the network, capturing the effects of bit rate adaptation, varying packet sizes, and traffic burstiness.

3. It has low overhead and causes no downtime: Being passive, PIE does not take away wireless bandwidth resources from users.

4. It does not require client modifications: The PIE mechanism is implemented at the APs and a central controller placed within the enterprise wired network. No client modifications are required.

PIE relies on the accurate timestamping of transmissions by the AP. These timestamps could be reported accurately by the firmware of the AP's wireless card. However, most off-the-shelf wireless cards do not expose this functionality and hence in our current implementation we use a second card at the AP to gather accurate timestamps of wireless transmissions.

Key contributions

This paper makes the following key contributions:

• We identify the key requirements for a practical interference estimation mechanism. We then carefully design PIE to meet those requirements and report various design choices to infer interference in real time.

• We evaluate the accuracy and agility of PIE using both controlled experiments as well as by playing back real traffic traces. For 95% of the links, PIE achieves accuracy comparable to the state-of-the-art technique of bandwidth tests (see §2). We further show that PIE can efficiently track the changing interference patterns caused by client mobility, variable transmission rates and varying traffic loads. Results from our playback of real traces indicate that PIE can converge to the correct interference estimate within 540 ms, 700 ms and 900 ms for heavy, medium and low traffic load periods. This represents up to 300× of speed up over bandwidth tests.

• *Demonstrate the utility of PIE in interference mitigation mechanisms:* We show the usefulness of PIE by integrating it with three interference mitigation mechanisms 1) Centralized scheduling, 2) Transmit power control and 3) Channel assignment. We show that real-time conflict information provided by PIE can enhance the performance of such mechanisms and outperform bandwidth tests under dynamic settings.

• *Employ PIE to uncover performance issues in two production WLANs:* We use PIE to monitor two production WLANs. We show that PIE can correctly infer subtle performance issues like asymmetric channel access and hidden terminal problems.

The rest of the paper is organized as follows. §2 discusses the current state of art in wireless interference estimation. The fundamental principles behind PIE are described in § 3. We present the design and operation of PIE in §4. We evaluate and validate our mechanism in §5. Finally we conclude in §6.

	Interference maps [15]	Microprobing [3]	CMAP [24]	PIE
No client mods	×	√	×	√
Online	×	√	√	√
Zero downtime	√	×	√	√
Real traffic	×	×	√	√
No wireless control traffic	×	×	×	√

Table 1: Comparing PIE with other interference estimation mechanisms.

2 Related work

We classify prior interference estimation and wireless monitoring efforts into the following categories.

Interference estimation tools : Bandwidth test mechanisms [16, 15] systematically transmit a simultaneous burst of traffic along each pair of AP-client links and observe how the aggregate throughput differs from the throughput achieved by each link operating in isolation. Recently, Ahmed et al. [3, 4] proposed the use of micro-experiments, each lasting less than a millisecond, to detect different kinds of conflict between WLAN nodes. Such mechanisms require network downtime and must rely on certain traffic pattern to test the interfering links, which may be deviant from real traffic scenarios.

CMAP [24] is a technique designed to solve exposed terminal problem using passive conflict graphs. However, it requires the interferers to be in the communication range of the receiver and will miss conflicts in which the interferer is outside the communication range but inside the interference range. Further, it requires driver level modifications to both APs and clients. Given that CMAP relies on modified clients, it is better able to infer uplink conflicts as well. However, since the fraction of uplink traffic might be limited (as reported for some enterprise WLANs [22]), we take the penalty of missing some uplink conflicts in order to avoid client modifications. Table 1 presents a comparison of our design of PIE with some prior proposed interference estimation tools.

Wireless monitoring studies: Researchers have recently conducted several studies to understand the performance of different 802.11 networks using trace collection, followed by empirical analysis. Each system is designed to analyze specific aspects of an 802.11 wireless network, ranging from physical and link-level behavior [21, 2, 24], client coverage [7], to understanding the performance of TCP/IP in wireless environments [9]. However, most of these mechanisms are geared towards offline analysis of wireless traces to derive interesting measures for their target 802.11 network. Recently, a short paper [6] proposed a machine learning approach to infer high-level interference. However, the proposed technique provides limited visibility and does not capture all types of interference.

Finally, WIT [13] and Jigsaw [8] are interesting measurement studies that have influenced some of the design decisions in PIE . In WIT, traces are captured using 5 sniffers in a wireless network and a state machine based learning approach is proposed to study the performance of the 802.11 MAC protocol in a practical deployment. Jigsaw deploys a large wireless monitoring infrastructure consisting of 150 sniffers to monitor a production WLAN and performs a cross-layer analysis to diagnose performance problems. Both these mechanisms present excellent insights into the functioning of a 802.11 network, but unlike PIE, they do not focus on evaluating the accuracy and agility of their interference estimation mechanisms, especially under interference settings that can arise due to client mobility and the use of bit rate adaption mechanisms. Also, they do not discuss the integration of their interference estimation mechanisms with applications like power control and channel assignment.

3 Interference estimation in PIE

Interference in an enterprise WLAN can be broadly classified into two categories: (a) sender-side interference caused by carrier sensing between two transmitters, and (b) receiver-side interference caused by collision at the receiver. While carrier sensing determines how the transmitters share the wireless medium, collision-induced interference determines whether transmissions are successfully decoded at the intended receiver. The goal of PIE is to identify both of these interference properties in a non-intrusive manner. We now explain the intuition behind PIE with the help of a simple example.

Intuition behind PIE: Consider a scenario from an enterprise WLAN (shown in Figure 1) where APs A and B are far enough apart such that they cannot carrier sense (CS) each other. Assume that two clients C_A and C_B are associated to APs A and B respectively. Suppose some downlink packets are being enqueued and being transmitted by APs A and B, for transmission to their respective clients, C_A and C_B. The APs follow the regular 802.11 carrier sensing mechanism, and transmit to their clients whenever possible.

In PIE, APs A and B periodically send their frame transmission timestamps to the controller. Further, the frames are tagged with their reception status indicating whether this frame transmission was successful or not (i.e., whether the AP has received an ACK for this frame or not). The controller parses these timestamps and identifies the four scenarios shown in Figure 1(b). Looking at scenarios 1 and 2, the controller observes that frame transmissions from A and B (denoted by P_A and P_B) overlap in both directions, indicating that A and B do not defer to each other, and hence are not within carrier sense range. Additionally, the controller can also infer that whenever a transmission for client C_B overlaps with a transmission by AP A, then C_B is not able to decode the transmission (i.e., P_B is lost). On the other hand, transmissions for C_A are not lost despite overlapping transmissions by AP B. Hence the controller concludes that AP A interferes with link (B, C_B) but B does not interfere with (A, C_A). The controller can then use this information to efficiently mitigate interference for C_B. For example, it can perform downlink data scheduling [22] and allocate different time slots to (A, C_A) and (B, C_B). Alternatively, the controller can also assign different channels to APs A and B, thereby allowing both transmissions to proceed simultaneously without any interference. As this example demonstrates, having accurate interference estimates could enable the controller to improve client performance in an enterprise WLAN by employing interference mitigation mechanisms effectively. We now give a detailed explanation of how PIE identifies these interference properties in a non-intrusive manner.

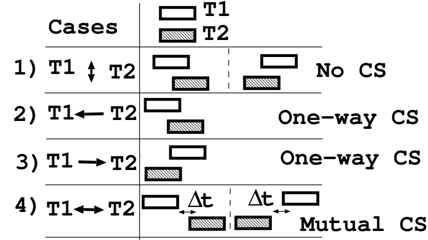


Figure 2: Detecting the carrier sense relationship between two links on the basis of timestamps of transmissions by the two transmitters A and B. Timestamps refer to the MAC timestamp of wireless frames as reported by the wireless card.

3.1 Estimating carrier sense (CS) interference

PIE identifies the carrier sense relationships based on the order in which competing transmitters access the wireless channel. Figure 2 shows the possible order of channel access for different carrier sensing relationships. As shown, there can be four cases of channel access:

(a) Overlapping frame transmissions (Cases 1, 2 and 3): Case 1) When two competing transmitters are not in carrier sensing range, they can access the channel in any order and hence the controller would observe that their frames overlap in both directions. Case 2,3) In case of one-way carrier sensing, the frames will only overlap in one direction. For example, if $T_1 \leftarrow T_2$ (i.e., T_1 carrier senses T_2) then T_1 will defer for T_2's transmissions. However, T_2 will not defer for T_1's transmissions, and would transmit even if T_1's frame is still in air. Hence the controller should only observe overlaps when T_1's transmission is already in the air and is overlapped by a later T_2 transmission.

(b) Non overlapping transmissions (Case 4): If both the transmitters can mutually carrier sense each other,

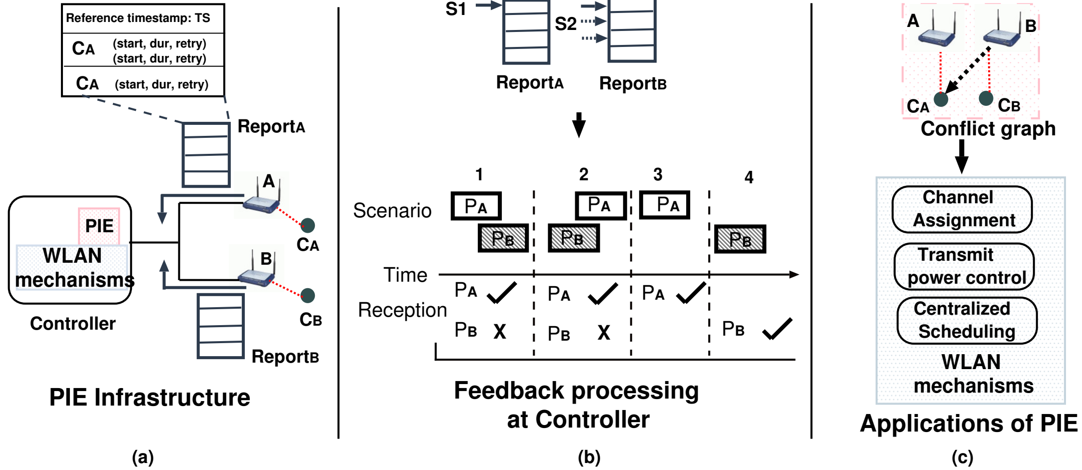


Figure 1: Overview of PIE, showing the overall infrastructure, the feedback processing performed at the Controller and the integration of PIE with channel assignment and scheduling. The detection of conflict between AP B and client C2A i) places the two APs in separate channels when channel assignment is performed, or ii) serializes the transmissions between AP A and B.

the controller should not see any overlaps as carrier sensing will serialize their frame transmissions. However, we note that non-overlapping transmissions may also be observed in scenarios where the two transmitters do not simultaneously contend for the channel, and transmit their frames one after another due to their specific traffic patterns. In such a scenario, it is difficult to make any inference regarding carrier sense relationship of the two transmitters. In order to distinguish the cases where transmitters are actually contending for the medium, we use the mechanism outlined in [13]. The controller labels a pair of frames as being transmitted by c̈ontending" transmitters if their starting timestamps are within a time interval γ, where γ is the total time that can be spent by competing transmitters performing back-off. Although all traffic within the γ interval may not contend for the channel, this heuristic was shown to be effective for practical settings [13]. We use a value of $\gamma = 28 + 320\mu s$ (DIFS + Max back-off period for 802.11g). The pseudo-code for estimating carrier sense properties in PIE is shown in Algorithm 1 (Procedure ComputeCS).

Algorithm 1 PIE : CS and INT computation

Procedure ComputeCS:
  Inputs: number of frames in contention n_c, number of case (3) overlaps n_f, and number of case (2) overlaps n_r, cs threshold δ_t (δ_t = 0.8 in our implementation)
  $n_o = n_f + n_r$
  $n_n = n_c - n_o$
  if $(\frac{n_n}{n_c} > \delta_t)$ **then**
    /* case 4 (A and B sense each other) */
    return $\frac{n_n}{n_c}$
  else if $(\frac{n_o}{n_c} > \delta_t)$ **then** /* sufficient overlaps to compute prob */
    if $(\frac{n_r}{n_c} > \delta_t)$ **then**
      /* cases 3 (A senses B) */
      return $\frac{n_r}{n_c}$
    else
      /* case 1 (A and B do not sense each other) */
      return $\frac{n_n}{n_c}$
  else
    /* inconclusive (wait for more samples) */
    return $-$
Procedure ComputeINT:
  Inputs: total number of frames n_p, number of frames lost n_l, number of overlapping frames n_o, number of overlapping frames lost n_{ol}, overlapping packets threshold β_t (β_t = 20 in our implementation)
  if $(n_o > \beta_t)$ **then**
    $l_{iso} = (n_l - n_{ol})/(n_p - n_o)$ /*loss in isolation*/
    $l_{int} = n_{ol}/n_o$ /*loss under interference */
    LIR = $(1 - l_{int})/(1 - l_{iso})$
    return LIR
  else
    /* inconclusive (wait for more samples) */
    return $(-)$

3.2 Estimating collision induced interference

PIE identifies collision-induced interference at the receiver by computing the probability of a frame loss at the receiver when it overlaps with a simultaneous transmission from a competing transmitter. Intuitively, the extent of interference is directly proportional to the probability of losing overlapping frames. Note that this allows PIE to maintain a continuous interference model, where the extent of interference can be any value between 0 and 1. Such a model is better suited for realistic environments where the binary model of interference may not suffice. On the basis of this observation, in PIE, we use the Link Interference Ratio (LIR) described below, as the metric to quantify interference for a link.

Link Interference Ratio (LIR): For a pair of interfering links, LIR captures the loss in performance observed when the two links are interfering, as opposed to operating in isolation. Consider a link (A, B) and its interferer C. We measure D_{AB}, the delivery probability of the link (A, B) in isolation (A is active, C is inactive). We then measure D^C_{AB}, the delivery probability of the link when

interferer C is also active with A. The LIR is given by:

$$LIR = D_{AB}^{C} / D_{AB} \quad (1)$$

LIR takes values between 0 and 1. LIR of 0 means that link (A, B) cannot deliver frames in the presence of C, while LIR of 1 means that C does not impact link (A, B). LIR values between 0 and 1 indicate the extent of interference on link (A, B) by interferer C. When A and C are in carrier sense range, LIR will be equal to 1, since the interferer C is able to share the channel with the transmitter A without causing any decrease in the delivery ratio of link (A, B) [1]. The pseudo-code for estimating interference is shown in Algorithm 1 (Procedure ComputeINT). PIE requires a certain threshold of overlap packets (β_t) to accurately estimate the loss rate under interference. We use $\beta_t = 40$ for our implementation as it is the smallest threshold that yields stable interference estimates under diverse experimental scenarios.

Handling simultaneous overlaps from multiple interferers: A client packet may overlap with multiple simultaneous transmissions from potential interferers. In such a scenario, the packet overlap and its subsequent loss or success is attributed to each overlapping interferer. Further transmission diversity will allow PIE to observe events that will distinguish the true interferer from the nodes that happened to transmit at the same time (future overlapping transmissions by false interferers will not lead to loss). As we show later in our evaluation in §5.1.3, there is significant diversity in wireless transmissions in realistic settings to allow PIE to operate efficiently in practice.

4 PIE Design and Operation

In this section, we describe the design and operation of PIE. A schematic overview of the overall design can be seen in Figure 1. PIE has the following three components.

Sniffing at the APs: In our current implementation of PIE sniffing of the wireless medium is limited to the APs in the enterprise WLAN. This allows us to avoid the additional overhead associated with the deployment and management of extra sniffers in the enterprise building. However, sniffing solely at the APs might result in reduced coverage of uplink client traffic, as compared to a dense sniffer deployment (e.g., as in Jigsaw [8]). In order to overcome this limitation, we employ the finite state mechanisms outlined in [13] (based on 802.11 states) to infer some of the missing client transmissions. We note that even with such mechanisms, it is difficult to capture all uplink client transmissions using monitors at the AP, and hence PIE may not be able to detect all uplink conflicts accurately. However, we accept this penalty of missing some uplink client conflicts in order to avoid deploying additional monitors.

PIE requires accurate timestamp information for accurate interference estimation. However, due to limitations of the existing Atheros driver and firmware, it is difficult to extract the exact time at which a packet is transmitted over the medium[2]. In order to overcome this problem, in our implementation of PIE, APs are equipped with two radios: one radio is used for normal packet transmissions and receptions, while the other radio is used for capturing packets on the wireless medium. The Atheros driver timestamps every frame that is received over the interface using an on-board 64-bit microsecond resolution timer. Thus a second radio that captures packets can record the exact timestamp of the packet transmission. Moreover, the proximity of the two radios ensures that the second radio receives the majority of frames transmitted by the AP due to capture effect.

Synchronization of clocks at the APs: PIE needs the APs to synchronize their clocks so that the controller can compare their packet transmission reports and determine the extent of overlap between any two transmissions reported by the APs. Further, time synchronization should be tight to allow accurate 802.11 analysis, on the order of 20-30 μs [8]. Prior mechanisms for 802.11 analysis [8, 9, 13, 26] synchronized the APs by finding common beacon packets in their transmission reports. However, performing such offline synchronization at the controller can be time consuming, and impractical for a real time interference estimation mechanism. To synchronize the clocks across the APs, we use the time synchronization protocol implemented by the Atheros driver [1]. As part of the protocol, the AP embeds a 64-bit microsecond granularity time stamp in every beacon frame, and the nodes that listen to the AP adjust their local clock based on this broadcast timestamp [12]. In order to make this synchronization seamless, we set up a virtual ad hoc interface on the second radio of each AP. Now all the APs that join the ad hoc network, synchronize themselves in real time using the beacons of the reference AP for the network. This approach has two key benefits:1) it is an online mechanism, meaning the nodes synchronize their clocks every time beacons are received from neighboring nodes, and, 2) it is transitive in nature, and works as long as the network is not partitioned.

[1] Note that this measure of LIR differs slightly from the interference metric proposed in [16], that relies on effective throughput and not delivery probability. However, throughput based LIR is ambiguous for carrier sensing scenarios, where a LIR value of 0.5 could mean 50% loss or carrier sensing. Hence we use delivery probability as it provides greater clarity into the LIR values in all scenarios.

[2] This is because once the driver passes the packet to the firmware, a variable delay is introduced based on the length of the firmware transmit queue and the amount of time the radio performs carrier sensing/back-off. Further, retry and other 802.11 packets (like beacons) are handled solely by the firmware, making timestamp estimation more challenging.

Section	Objective	Topology	Observation
§ 5.1.1	Accuracy of PIE for canonical topology	2-link (Hidden / Exposed / Normal) topology	PIE is accurate within ± 0.1 of ground truth for 95% of scenarios
§ 5.1.2	Accuracy of PIE under client mobility, variable bit rates and packet sizes	2-link (Hidden) topology	PIE is able to track the changing interference patterns in real-time ($\sim$ 100 ms)
§ 5.1.3	Evaluate accuracy with multiple simultaneous transmitters	15-node topology	PIE is accurate when transmitters overlap less than 75% of time
§ 5.2.2,5.3	Convergence time of PIE under real trace-based traffic replay	15-node topology	Median convergence time is 400, 600, 720 ms for heavy, medium and light client traffic
§ 5.4.1,5.4.2 § 5.4.3	Performance of channel assignment, power control & scheduling with PIE	15-node topology	Outperforms bandwidth tests in dynamic cases ($1.25\times$, $1.50\times$ gain in goodput, fairness)
§ 5.4.4	Performance diagnostics in two production WLANs	386 & 464 AP-client links	8-11% links suffer from hidden terminals and 20% links show rate anomaly problems

Table 2: Summary of evaluation results.

Collecting and processing feedback from the APs: In PIE the Controller periodically polls the APs for their transmission reports. The granularity of polling is a tunable parameter, which can be determined empirically. Lower polling periods will enable PIE to update interference estimates faster. On the other hand, increasing the polling period allows APs to sample more packets per transmission report, increasing the accuracy of interference estimates. We evaluate this tradeoff in §5 and show that a polling period of at least $\sim$100 ms is needed to achieve good accuracy for PIE . Feedback processing at the Controller takes O(m^2n) time, where m is the number of APs and n is the number of packets per AP[3].

Handling multi-rate links: The exact impact of an interferer on a transmitter-receiver pair also depends on the physical layer bit rate being used by the transmitter. PIE tags the LIR value for each link-interferer pair with the bit rate being used for packet transmission on the wireless link. During the computation of LIR values as described in §3.2, overlap and isolation losses are recorded separately for each physical layer data rate and then the corresponding LIR value is computed for each rate. The Controller maintains a two-level lookup table for LIR values, where the first level is indexed by the link-interferer pair and the second level provides values for different rates used by the link for the given interferer. This data structure can also be extended for tagging conflicts with the transmit power level of the interferer, allowing the Controller to estimate the level of conflict under different power levels.

Interaction with external interference: External interference can be caused by non-enterprise wireless traffic and/or non-WiFi traffic (like microwaves). In the first case, if the non-enterprise traffic source is visible to any enterprise AP, its transmission timestamps would be reported to the PIE controller, which could then use the normal procedure to detect if the external source is causing any problems for the enterprise clients. In the second case, when the external interferer is not visible (like a non-WiFi source or a hidden external WiFi source) to any enterprise AP, PIE would not be able to identify the source of interference.

[3]Since the transmission report by each AP is already sorted, the overhead of merging at the Controller is small.

5 Evaluation of PIE

We divide the evaluation section into three distinct subsections. First, we demonstrate that PIE accurately captures interference in real time. We do so by comparing PIE with bandwidth tests. Next, we measure the time taken by PIE to converge to accurate interference estimates, under both controlled traffic loads and realistic trace replay on the wireless testbed. Lastly, we integrate PIE with a number of real time WLAN optimization mechanisms to offer evidence that PIE is useful for real-time problem diagnosis on a WLAN.

We evaluate PIE on two different testbeds. We run our central controller on a standard Linux PC (3.33 GHz dual core Pentium IV, 2 GB DRAM) (in about 3,000 lines of C code and a few hundred lines of Perl script), and Soekris (Testbed 1) as well as VIA-based (Testbed 2) wireless APs, modified slightly to improve path latencies. Each node in the two testbeds is equipped with two Atheros AR5212 chipset wireless NICs. We use saturated UDP traffic for our experiments unless otherwise specified.

Summary: A summary of the results presented in this section is shown in Table 2. Our results show that (i) PIE accurately estimates LIR under different carrier sensing and interference relationships, (ii) PIE can handle client mobility, variable bit rates and packet sizes, (iii) PIE is able to distinguish between multiple interferers when overlap in transmissions is less than 75%, (iv) PIE converges within 100 ms for saturated traffic, and within 400 ms, 600 ms and 720 ms when heavy, medium and light activity traffic periods are replayed from a real trace, (v) PIE enables WLAN applications to perform efficiently in dynamic scenarios, (vi) PIE can identify performance problems in hidden terminals and rate anomaly in production WLANs.

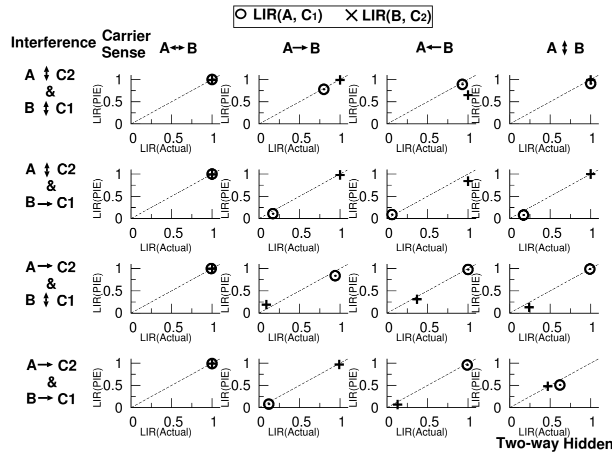


Figure 3: Scatter plots comparing the LIR values of PIE with the ground truth computed using unicast bandwidth test for all possible combinations for carrier sensing and interference relationships that can occur in a two link canonical topology. Packet size and data rate was fixed at 1400 bytes and 6M respectively. Note that for all scenarios, the value computed by PIE is close to the value reported by bandwidth test, as indicated by the proximity of these values to the x=y line in the plots.

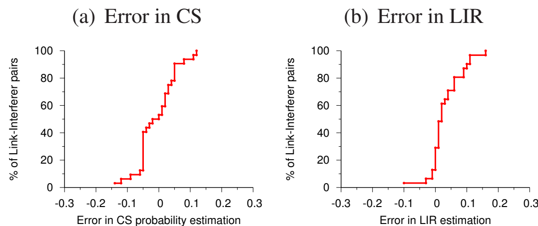


Figure 4: Distribution of error in predicting (a) Carrier Sense probability, and (b) LIR value as compared to the ground truth computed using unicast bandwidth tests, for the sixteen canonical scenarios outlined in Figure 3.

5.1 Accuracy of PIE

We evaluate PIE's accuracy using two different methods. First, we construct all possible conflict scenarios using a canonical two link topology. This experiment serves as our controlled experiment that allows us to assess accuracy and focus on the underlying phenomena causing any discrepancies between PIE and bandwidth tests. Second, we generalize our findings across a large-scale testbed, quantifying PIE's overall accuracy. Overall accuracy is further evaluated across a number of dimensions that take into account diverse transmission rates, packets sizes, interference scenarios, and density.

Metrics for comparison: Both experiments are evaluated according to the Link Interference Ratio (LIR) described in §3.2. LIR is the ratio of the frame delivery probability [4] of a link (A, B) under interference from C and in isolation (D^C_{AB}/D_{AB}).

Compared schemes: We compare three approaches that measure LIR with differing levels of overhead.

1) Unicast bandwidth tests (Ground truth): The conventional approach, is to use unicast bandwidth tests (UBT) to determine the impact of an interferer on a link [16]. In unicast bandwidth tests, A transmits unicast packets to B in isolation and under interference from C. We then report LIR as the ratio of frame delivery probabilities under the two scenarios. This is an accurate test to determine LIR as it uses unicast traffic, which takes into account the impact of C on the receiver (data packet collisions) and the sender (ack collisions). Henceforth, we use the LIR value reported by unicast bandwidth tests as the g̈round truth" in our experiments. Note that UBT incurs significant overhead – it takes O(n^4) measurements to compute a conflict graph for a n node topology, and hence is not practical to use under dynamic wireless environments.

2) Broadcast bandwidth tests : In broadcast bandwidth tests (BBT), broadcast traffic from A to B is used to compute the frame delivery ratios, both in isolation and under interference from C. This method was proposed as a relatively fast way to measure interference relationships among a large number of links [16]. Broadcast tests can compute the conflict graph for a topology of n nodes using O(n^2) measurements (as opposed to O(n^4) for UBT). However, broadcast tests do not take data-ack collisions into account and hence may be inaccurate in some scenarios.

3) PIE : PIE computes the LIR value in a passive fashion by determining the conditional loss probability of packets on link (A, B) that are interfered by interferer C. A packet P_i on link (A, B) is considered to be interfered if it overlaps with a transmission from interferer C that leads to packet loss. The LIR in this case is computed by passively observing the events in the wireless medium as recorded at the controller. Psuedocode for PIE is shown in Algorithm 1 (function ComputeINT).

In what follows, all experiments are performed using 802.11a (except the live WLAN measurements in §5.4.4), to prevent interference from the co-located department WLAN that uses 802.11g. Furthermore, the PIE measurements are collected passively through the observation of the probe traffic generated by the bandwidth tests.

5.1.1 Static interference settings

We start by comparing the LIR generated by the three mechanisms for different canonical scenarios, as shown in Figure 3. In order to have a fair comparison, we first evaluate the accuracy of PIE under static data rate (6Mbps) and packet size (1400 bytes) settings, as the overhead for computing LIR for dynamic (client mobility, variable rates) can be significant for bandwidth tests. We then relax these constraints and evaluate the performance of PIE under dynamic interference scenarios triggered by client mobility, the use of variable transmission rates and different packet sizes.

[4] 802.11 ACK is included into frame delivery rate for unicast frames

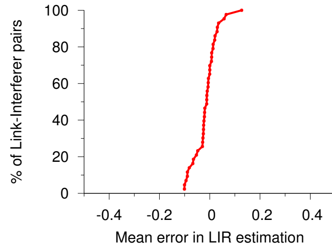


Figure 5: Distribution of error for PIE as compared to LIR values computed using UBT. We note that in 95% of the interference scenarios PIE is within 0.1 of the actual LIR value.

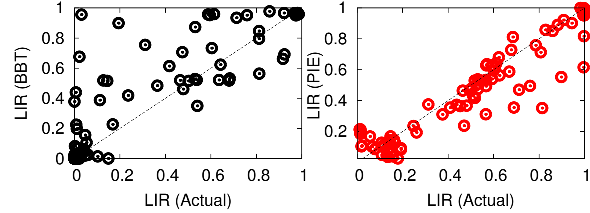


Figure 6: Scatter plot of delivery ratios obtained using bandwidth tests (unicast - LIR(Actual), broadcast - LIR(BBT)) and PIE on 43 link pairs. Note that LIR(BBT) may underestimate the loss rates as it does not take the ACK loss into account.

Controlled experiments: Using a canonical two link topology we benchmark different carrier sensing and interference scenarios. We selectively disable the carrier sensing of transmitters to create the complete set of scenarios. The possible interference relationship between the two links assuming that C_1 is associated with AP A, and C_2 is associated with AP B are as follows: (i) A interferes with C_2 and B interferes with C_1 ($A \rightarrow C_2 \wedge B \rightarrow C_1$), (ii) A interferes with C_2, B does not with C_1 ($A \rightarrow C_2 \wedge B \updownarrow C_1$), (iii) B interferes with C_1, A does not with C_2 ($A \updownarrow C_2 \wedge B \rightarrow C_1$), and (iv) A, and B do not interfere with each others client ($A \updownarrow C_2 \wedge B \updownarrow C_1$). Further, the possible carrier sensing relationship between the two transmitters are: (i) A and B carrier sense each other ($A \leftrightarrow B$), (ii) B carrier senses A ($A \rightarrow B$), (iii) A carrier senses B ($A \leftarrow B$), and (iv) A and B are not in carrier sensing range ($A \updownarrow B$).

Figure 3 compares the LIR values computed by PIE and unicast bandwidth test for the sixteen possible scenarios of carrier sensing and interference between two links. It also identifies cases which correspond to mutual (two-way) and asymmetric (one-way) hidden terminals. As shown in the figure, the LIR estimates of PIE are very close to the values reported by the unicast bandwidth tests. Also, Figure 4 shows the distribution of error in estimating carrier sense probability and LIR values for these different scenarios. As clear from the figure, PIE is able to estimate the carrier sensing and LIR values with good accuracy (± 0.15) for all scenarios. Note that identifying both carrier sensing and LIR values accurately can characterize client performance under any scenario. For instance, in the scenario where the interference relationship is $A \rightarrow C_2 \wedge B \rightarrow C_1$, the links can achieve similar throughputs when they are carrier sensing and sharing the channel ($A \leftrightarrow B$) or when they are not carrier sensing (two-way hidden terminal) and there is close to 40% loss rate for the links. PIE can provide this greater visibility, as to which phenomenon is actually taking place, which can then be used by interference mitigation mechanisms.

Accuracy in larger testbed: We repeat the experiments reported in Figure 3 for a large number of link pairs in our testbed, comprising 30 nodes spread across five floors of our department building. We select links whose delivery ratio in isolation is greater than 0.9 in both directions [3][5]. Figure 5 compares the values of LIR achieved using unicast bandwidth test and PIE for 43 interference scenarios. We note that for 95% of the interference scenarios, PIE is within 0.1 of the actual LIR value. We experimented with different convergence thresholds and found that convergence within 0.1 of the actual LIR value was sufficient for practical applications (see §5.4 for performance of such applications).

Finally, we note some inaccuracies that are introduced through approaches like BBT, which aim to collect interference information at low overhead. BBT will mis-estimate when interference impacts the reception of ACKs rather than data packets. Figure 6 does indeed confirm that such cases do exist in reality and that they lead to the underestimation of loss.

5.1.2 Dynamic interference settings

The previous experiments quantified PIE's accuracy as compared to the ground truth generated using unicast bandwidth tests. However, PIE is not only able to accurately capture interference under static conditions, but more importantly, also under dynamic conditions.

Handling client mobility: Any practical interference estimation mechanism must be able to handle client mobility, i.e. it should be able to update the conflict graph in real time to reflect the changing interference patterns that arise due to client movement. In order to evaluate PIE 's ability to handle mobile clients, we perform a micro experiment, where a mobile client is moving away from its AP towards a hidden interferer as shown in Figure 7. In this experiment, the client is moving at a pace of 0.25 m/s [6]. The bottom plot in the Figure shows the signal strength at the client from the AP and the interferer, while the middle and top plots show the throughput of the mobile client and the LIR estimate by PIE at each instant in the experiment. As shown in the Figure, PIE's LIR estimate decreases as the client moves towards the interferer. Furthermore, it closely matches the trend shown

[5] We wanted to consider stable links (high SNR) for analysis. In reality, poor SNR links would rarely be selected during client association to APs.

[6] Normal walking speed for mobile user.

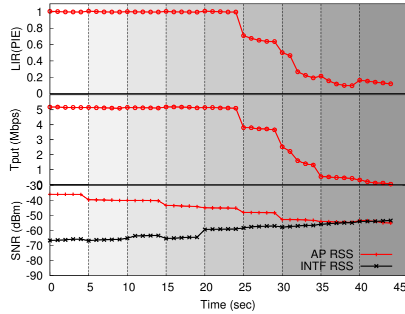


Figure 7: PIE 's ability to track the changing interference patterns for a mobile client. In this experiment, a mobile client is moving away from its AP towards a hidden interferer. The bottom plot shows the signal strength at the client from the AP and the interferer. The middle plot shows the throughput achieved by the client at each instant. The top plot shows the LIR as measured by PIE.

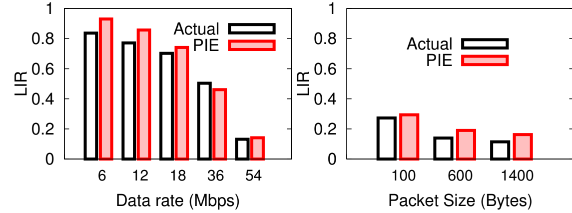


Figure 8: Impact of physical layer data rate and packet size on the delivery ratio of a link in a canonical hidden terminal topology. While varying data rate, packet size is fixed at 1400 bytes, and while varying packet size, data rate is fixed at 24Mbps. Note the significant drop in delivery ratio with rate while the impact of packet size is less pronounced. 90% Confidence intervals were found to be tight and hence are omitted for clarity.

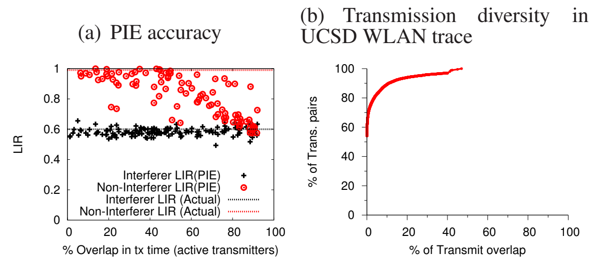


Figure 9: Ability of PIE to identify true interferers from a set of active transmitters. (a) LIR measured by PIE for both the true interferer and the non-interfering transmitter as a function of the overlap in transmission times. If the overlap fraction is less than 75%, PIE can distinguish the false and true interferers accurately. (b) Overlap in transmission times for all wireless transmitter pairs that are active during a one hour time window (2pm - 3pm) in the USCD wireless trace. As clear from the trace, about 90% of the transmitter pairs overlap less than 20% of the times, providing sufficient traffic diversity for PIE.

by the instantaneous throughput during the experiment, which confirms PIE's accuracy in predicting the end user performance in dynamic wireless environments.

Variable rate and packet sizes Prior research [24, 5] has shown that the interference properties of wireless links are impacted by the data transmission rate and packet size. In order to evaluate PIE for different packet sizes and data rates, we repeat our canonical experiments with different packet sizes and data rates on multiple interferer-link pairs. To evaluate multiple data rates, we first activate a link in isolation and then activate an interferer, which forces the transmitter to adjust its data rate to minimize losses. We use the default Atheros rate adaptation algorithm, SampleRate. Figure 8 (left) shows the impact of data rate on the delivery ratio of a link (LIR by UBT) and the estimate of LIR generated by PIE for each rate in the experiment.

Next, we fix the data rate and vary the packet size for a link under interference (right plot). As expected, LIR is worse for larger packet sizes, which are prone to more errors. We observe that the combination of data rate and packet size can result in varying interference properties and PIE is able to efficiently identify the impact of interference accurately in each such scenario (confirmed by the agreement with UBT). This also shows that using bandwidth tests or other active measurements may require performing an exponential number of tests with varying packet sizes and rates to determine the interference impact for any given traffic scenario. PIE, on the other hand, can passively determine the extent of interference for each scenario efficiently and accurately.

5.1.3 Classifying interferers accurately

PIE's fundamental operation relies on observing overlap in transmissions and correlating such events with packet loss. One could argue that PIE's accuracy is likely to be affected by scale since the probability of observing overlap in transmissions across the network increases with greater scale. Then the probability of identifying the transmitter responsible for loss becomes much harder. To answer this question we attempt to quantify the success of PIE in correctly identifying an interferer depending on the amount of time that it tends to overlap with the transmitter suffering the loss.

Canonical experiments: Consider a link (A, B) and two interferers C_1 and C_2. We compute the actual LIR of the link under C_1 and C_2 by performing individual unicast bandwidth tests, first with C_1 and then with C_2. According to the unicast tests, the LIR of the link under interference from C_1 and C_2 is 0.6 and 0.99 respectively, indicating substantial interference from C_1 and no interference from C_2. We term C_1 as the interfering transmitter and C_2 as the non-interfering transmitter. Our goal is to evaluate the accuracy of PIE in identifying the interfering (C_1) and non-interfering (C_2) transmitters, when both C_1 and C_2 are activated simultaneously. Both C_1 and C_2 follow a http traffic model, with sleep and active times being drawn from a 802.11 wireless trace [13]. We then identify the time periods (1s) in the experiment with varying overlaps between the transmission times of C_1 and C_2 and measure the LIR values for C_1 and C_2 according to PIE.

Figure 9(a) shows the LIR obtained by PIE for both the interfering (C_1) and non-interfering (C_2) transmitter as a function of the overlap in their wireless transmission times. As expected, when the overlap in transmission times is close to 100%, PIE is unable to distinguish between true and false interferers. When the overlap is less than 60% PIE can distinguish between the false and true interferer. In fact, notice that even for high overlaps (close to 75%), the median loss probability for false interferer is close to 0. Further, as shown in Figure 9(b) more than 90% of the transmitters in a real WLAN trace (UCSD WLAN [8]) overlap less than 20% of the time, indicating rich diversity in transmission patterns for wireless users. Such diversity will enable PIE to function efficiently in realistic deployments.

Multiple interferer experiments: To validate the previous result with multiple interferers, we repeat the aforementioned experiments in a larger topology. In our experiments, we try to emulate the structure of our in-building WLAN by placing one testbed AP node near each production AP in the environment. We present results from a representative topology that randomly distributes client nodes into offices. The topology has 7 APs and 8 clients. Clients connect to the AP with the strongest signal strength. Each transmitter follows a http on-off model for transmitting data with the on and off times derived from the UCSD trace. We classify all interferers for which the UBT LIR is less than 0.8 ($>$ 20% loss) as strong (interfering) transmitters and the rest are classified as weak (non-interfering) transmitters.

Figure 10 (a) shows the number of strong and weak interferers per client as determined by UBT in our topology. Figure 10 (b) shows the ability of PIE to identify multiple strong and weak interferers in this topology. As shown in the Figure, the LIR values estimated by PIE are within +/- 0.15 of the actual LIR determined by pairwise bandwidth tests using unicast traffic (UBT). Summarizing, PIE is able to accurately identify the exact impact of each interferer on every client in the system even in the presence of multiple simultaneous transmitters. We show the overall impact of such an accurate conflict graph on application level performance for wireless clients in the system in §5.4.

5.2 Agility of PIE

PIE can be integrated in today's centralized WLANs, requiring software-only modifications to the central controller. However, as is apparent from the design section, there are a number of knobs in PIE 's design that are likely to affect its accuracy. In this section, we study appropriate values for the polling interval, and measure PIE's convergence time under varying loads.

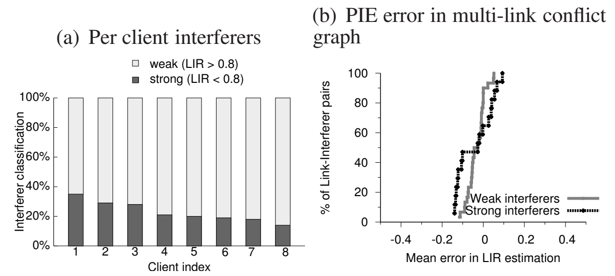


Figure 10: Accuracy of PIE for a 8 client, 7 AP topology. (a) Distribution of strong (LIR $<$ 0.8) and weak (LIR $>$ 0.8) interferers. (b) CDF shows the error in PIEś estimation of LIR for a link-interferer pair as compared to pairwise bandwidth test (UBT). PIE identifies both multiple strong and weak interferers accurately (all estimates are withing +/- 0.15 of UBT LIR values). PIE is able to identify the extent of interference accurately in the presence of multiple strong and weak interferers.

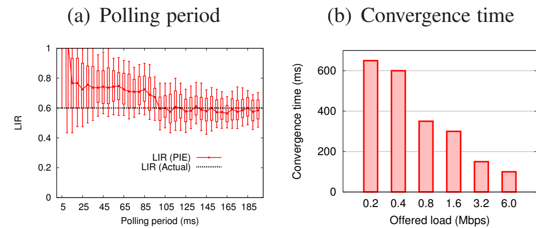


Figure 11: (a) Impact of polling period on the accuracy of the interference measures produced by PIE . LIR value stabilizes for polling periods greater than 100ms. The experiment time was adjusted to ensure same sample size for different polling periods. (b) Convergence time for a canonical hidden terminal link as a function of traffic load on the link and the interferer.

5.2.1 Polling interval

Any online interference estimation mechanism must identify conflicts in real time to be useful. In PIE , the controller periodically polls the APs for transmission summaries and then determines link conflicts. Higher polling periods can provide more information to the controller, thereby improving the quality of interference estimation. However, having a higher polling period also makes the system less responsive, which may be critical to dynamic interference scenarios. Here we evaluate the performance of PIE with different polling periods and determine the minimum period for which PIE can provide stable LIR values. We define a LIR value reported by PIE to be stable when the 90th and 10th percentiles of the LIR estimates differ by less than 0.1 of the mean LIR value. Figure 11 (a) demonstrates that a value of 100 ms provides a good compromise between reactivity and accuracy.

Note that smaller polling periods will also increase the communication overheads for sending traffic reports from the AP to the Controller. Using an average packet size of 600 bytes, and a medium constantly busy at 54 Mbps, the AP in PIE will have to store a summary for 1125 packets. This results in 9 KBytes sent from each AP every 100 ms, i.e. 1 Mbps, easily sustained by the AP.

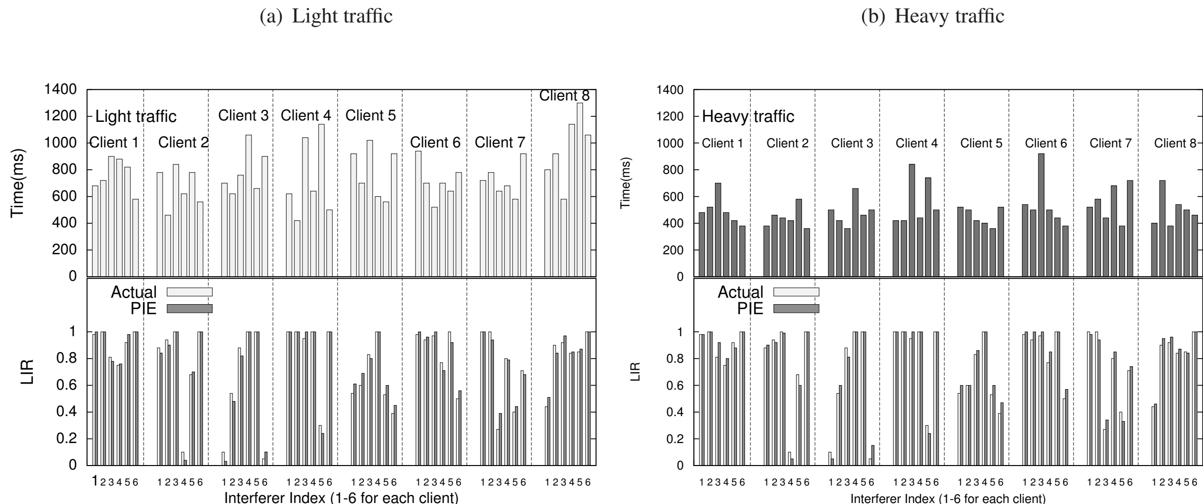


Figure 12: Convergence time and accuracy for PIE on a 7 AP - 8 Client topology under realistic patterns replayed from a period of (a) light client activity and (b) heavy client activity (using TCP). Top part of both figures shows the convergence time for each link-interferer pair and the bottom figure shows its corresponding accuracy when traffic traces are replayed on our representative topology. As shown in the figure, for light (heavy) traffic scenarios, PIE takes 1150ms (650ms) or less for 95% link-interferer pairs to converge within $\pm$ 0.1 of their actual value.

5.2.2 Convergence time

Convergence time is defined as the amount of time taken by PIE to gather sufficient samples to compute an accurate LIR estimate (within ± 0.1 of ground truth). Accordingly, the time taken by PIE to converge on an accurate estimate for link interference depends on two key factors: i) the polling period used by PIE to collect statistics from the APs, and ii) the actual amount of traffic that is captured by the APs in a given polling period. We first understand the impact of traffic load on the convergence of PIE by systematically varying the load on the canonical two link topology. Figure 11(b) shows the convergence time for a canonical hidden terminal link as a function of traffic load on the link and the interferer. Both the link and the interferer use a physical data rate of 6Mbps, while the traffic load is varied from 6Mbps (saturated) to 0.2 Mbps (light). Reduction in traffic load leads to longer convergence times because of the reduced frequency of interference events. Note, however, that LIR values would correspond to perceived client performance degradation only under relatively heavy loads, in which case PIE could capture events in 100 ms. In contrast, the measurement overheads of prior bandwidth test based active interference estimation mechanisms (e.g. Interference-maps [15]) is in the range of 20-30 seconds per link-pair [16].

Next, in order to understand the convergence of PIE under realistic traffic patterns, we replay a real WLAN trace [18] on the representative (7AP - 8 Client) topology (described in Section 5.1).

5.3 Experiments with real wireless traces

We now present experimental results on the performance of PIE using the publicly available Sigcomm 2004 traffic traces [18]. The Sigcomm trace was partitioned into heavy, medium, and light periods corresponding to periods with airtime utilization of more than 50%, between 20-50%, and less than 20% respectively, at different times of the conference [19]. In these traces, HTTP transactions were categorized into a series of HTTP sessions. Each session consists of a set of timestamped operations starting with a connect, followed by a series of sends and receives (called transactions), and finally a close. The HTTP sessions are then replayed on our testbed using the mechanism described in [10]. In our experiments, each client emulated the behavior of one real client from the trace, faithfully imitating its HTTP transactions. We use TCP as the underlying transport protocol for trace replay.

Figure 12 shows the convergence time (top plot) and accuracy (bottom plot) of PIE for each link-interferer pair when access patterns from the light and heavy load periods are replayed on the representative topology. As shown in the figure, for light (heavy) trafc scenarios, PIE converges to $\pm$ 0.1 of the actual LIR value within 1150 ms (800 ms) for more than 95% of the link-interferer pairs [7] Further figure 13 shows the distribution of convergence time of PIE for different link-interferer pairs under all three load periods. As expected, the convergence time is smaller for higher activity periods. The median convergence time for the light, medium, and heavy traffic loads are 400 ms, 620 ms, and 700 ms respectively.

[7] We skip detailed results from medium activity periods and instead show only the distribution for medium activity period to save space.

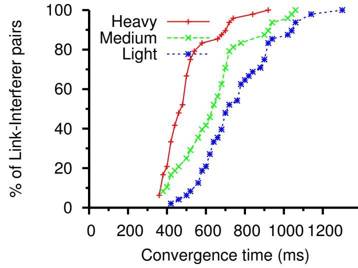


Figure 13: Distribution of convergence time for all link-interferer pairs under realistic traffic scenarios. Traffic scenarios (TCP based) are classified as heavy, medium and light depending on the total traffic load. As expected, PIE's convergence is faster for heavy traffic scenarios (median of 400 ms), followed by medium (median of 620 ms) and light (median of 700) traffic.

5.4 Applying PIE to improve WLAN performance

Being able to track interference in a highly dynamic environment may be considered as an admirable academic exercise. In this section, we will prove that access to such information can better enable a number of real time mechanisms that have been proposed for the performance optimization of wireless networks. To that end, we have integrated PIE with three such mechanisms (channel selection, dynamic packet scheduling, and power control) and tested them on two different testbeds. Our results clearly demonstrate that all these functions become a viable tool in the hands of network operators as long as we can supply reliable interference information in real time.

We use the same 7 AP and 8 client topology that we described in §5. We set the polling period to 1 second as per our observation in §5.2.2, thus capturing interference accurately even under low traffic loads. In mobility experiments, each client moves along a corridor at ∼0.25 m/s. We use UDP traffic for our experiments to measure the performance of PIE with different applications. We also perform experiments with TCP traffic for centralized scheduling application and report the results for the same.

Conflict graph	Mechanism (Num Channels)	System Tput(Mbps)	Jain's Fairness Index
NA	Single (1)	9.2	0.52
NA	LCCS (3)	17.1	0.58
UBT	Conflict aware (3)	24.6	0.72
PIE	**Conflict aware (3)**	**24.9**	**0.71**

Table 3: Performance of conflict-aware channel assignment (using conflict graph generated by PIE and bandwidth tests) as compared with single channel and LCCS (least congested channel search) assignments. Under static conditions, PIE leads to similar results as UBT, offering significant improvement compared to single channel and LCCS assignments. Note that UBT being an active technique has significantly higher measurement overhead and is not practical.

5.4.1 Application I: Channel assignment

Efficiently assigning channels to access points (APs) in an enterprise WLAN can significantly affect the network performance and capacity [14, 20]. We implement a conflict aware channel assignment heuristic (Randomized Compaction), proposed in [20], that takes a conflict graph as input and performs channel assignment with the objective to minimize interference. We compare the performance of the conflict-aware channel assignment scheme when based on the conflict graph generated by PIE and that of unicast bandwidth tests.

Table 3 shows the total system throughput and Jain's fairness index achieved by each channel assignment mechanism. Bandwidth tests are performed with unicast traffic at data rate and packet size of 6Mbps and 1400 bytes. Experiments are performed under static settings for a fair comparison with bandwidth tests. We consider the conflict graph generated by bandwidth tests as the true interference information. Results are averaged over 20 runs. We note that conflict aware channel assignment significantly improves system throughput over LCCS [11] (least congested channel search) and single channel assignments. Moreover, the performance of the heuristic is similar with PIE and bandwidth tests, illustrating PIE's ability to generate high quality conflict graphs in real time.

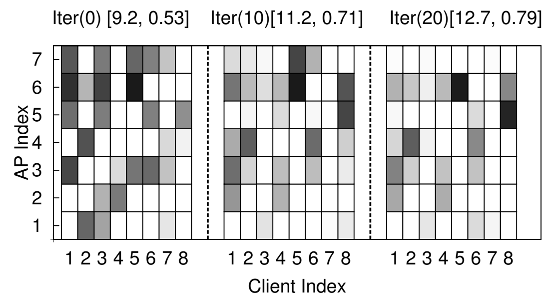


Figure 14: Performance of an iterative power control mechanism that uses PIE. Each matrix represents the conflict graph, with overall capacity (total system throughput in Mbps) and Jain's fairness index listed in the title. Intensity of darkness is proportional to the extent of interference. The final state corresponds to reduced interference, improved overall network capacity and fairness.

5.4.2 Application II: Transmit Power Control

We implement a simple centralized power control heuristic that uses the dynamic conflict information produced by PIE to reduce interference in the system. We measure the performance of the system through LIR_{all}, i.e. the sum of LIR values, for all link-interferer pairs in the system. Our goal is then to maximize this value by iterating over different power levels of the transmitters.

In each iteration of power control, we identify the most dominant interferer, as the AP that sources links with the minimum cumulative LIR. We reduce its transmit power

(by 10mW) and recompute the conflict graph using PIE. If the new conflict graph has lower cumulative LIR, then we discard the new power settings and reduce the power level of the next strongest interferer. In this way, we always move to a new set of power levels only if it increases the overall performance of the system. We quit when there is no improvement in the overall LIR value for 10 iterations.

Figure 14 shows the impact of such a power control mechanism. We present three matrices that capture the interference caused by each AP (row) to each client (column) in the network (the darker the cell, the stronger the interference). The title of each matrix further captures the iteration, the overall network capacity, and the fairness index. The leftmost matrix corresponds to the default power level setting, while the middle and right columns indicate the intermediate and final stages of the power level settings achieved by the aforementioned power control heuristic. We clearly see that our simple power control mechanism reduces the overall conflict in the system (matrix cells get increasingly lighter), while increasing overall network capacity and fairness. The point of this evaluation is not on the power control mechanism itself, since there are a number of solutions that could achieve such an objective more effectively (like [17]). Our focus is to demonstrate the effectiveness of PIE when used for power control.

5.4.3 Application III: Centralized scheduling

Accurate, fast and scalable conflict graph construction is critical for realizing centralized data plane mechanisms. In a recent work on centralized data path scheduling (Centaur [22]), authors relied on micro-probing [3], an online mechanism that performs micro experiments to determine link conflicts. Although micro-probing can generate an accurate conflict graph in very short time scales (4 seconds for a 10 link topology), it may still be inefficient in high mobility scenarios, especially given the need for silencing the network during the measurement of the conflict graph. We re-evaluate the performance of Centaur using the conflict graph generated by PIE and contrast it to bandwidth tests for consistency. We show that PIE improves the performance of Centaur under high mobility and varying traffic properties (variable packet sizes and data rates).

Table 4 shows the Centaur's performance when operating on conflict information from PIE and bandwidth tests respectively, in one static and one mobile scenario. The UBT conflict graph is generated using 6 Mbps and a fixed packet size of 1400 bytes for static client locations. Due to the overhead of recomputing bandwidth tests, we use the static conflict graph for the mobility scenario too. One can clearly see that exploiting real time conflict information in scheduling is not only increasing

Scenario	Mechanism	System Tput(Mbps)	Jain's Fairness Index
Static(UDP)	DCF	11.2	0.64
	Centaur (UBT)	12.6	0.88
	Centaur (PIE)	**13.0**	**0.84**
Static(TCP)	DCF	9.5	0.60
	Centaur (UBT)	12.2	0.85
	Centaur (PIE)	**12.4**	**0.89**
Mobile(UDP)	DCF	10.1	0.61
	Centaur (UBT)	10.4	0.71
	Centaur (PIE)	**12.4**	**0.95**

Table 4: Performance of centralized scheduling (Centaur) using PIE 's conflict graph. UBT and PIE lead to equivalent performance under static settings. The introduction of mobility confirms PIE's superiority to provide real time information. Note that UBT has very high measurement overheads compared to PIE .

the overall network throughput but also the fairness index across clients. More interestingly, the inaccuracies in the conflict graph generated using bandwidth tests almost negate the benefits of centralized scheduling under mobility. We performed similar experiments with auto-rate and observe that Centaur with PIE 's conflict graph provides 32% overall system throughput gain as compared to using the conflict graph generated using bandwidth tests under static scenarios (6Mbps, 1400 bytes).

TCP performance: We also analyze TCP performance for different conflict graphs. We observe system throughputs (fairness) of 9.5 Mbps (0.60), 12.2 Mbps (0.85) and 12.4 Mbps (0.89) for DCF, Centaur(UBT) and Centaur(PIE) respectively. As expected UBT and PIE perform close to each other and outperform DCF. However, as noted earlier, the measurement overhead of UBT is much higher than PIE making it impractical for real time mechanisms like Centaur.

5.4.4 Application IV: Wireless troubleshooting

Beyond PIE's ability to enable real time performance optimization in enterprise WLANs, its real time nature allows it to serve as a diagnosis tool that could be used proactively by a network operator to avoid performance problems. We test this property by running PIE in two production 802.11b/g WLANs (W_1 and W_2), co-located with our two testbeds.

These WLANs differ from each other in many significant ways as follows. $WLAN_1$ spans 5 floors of a building and uses 9 APs manufactured by vendor A. The network administrator was responsible for conducting RF site surveys, identifying locations to place the APs, and manually assigning the channel of operation of each AP to minimize interference. Exactly 3 APs were placed on channels 1, 6, and 11 in $WLAN_1$ to minimize the level of inter-AP interference. In contrast, $WLAN_2$ occupies a single floor of a different building, uses 21 APs manufactured by a different vendor, B, and features a controller in charge of dynamic channel assignment. The number of APs on each channel, thus, varies over time. In $WLAN_2$ the vendor was responsible for conducting the RF site surveys and making AP placement decisions.

WLAN	HT-Links ($LIR < 0.7$)	Anomaly-Link pairs ($Ratio < 0.2$)
WLAN1	31 / 386	231 / 1087
WLAN2	53 / 464	305 / 1391

Table 5: Performance issues observed in two production WLANs. The extent of hidden terminal interference ranges from 8% to 11% but can be significant for a small number of links. Rate anomaly affects approximately 20% of the links in both networks.

We select testbed nodes closest to the production APs to provide transmission reports to the PIE controller, sniffing the transmissions on the operational network. We use those reports to measure the carrier sense and interference relationships between different links in the production WLAN. PIE reveals two performance issues:

1) Hidden terminals: Performance degradation beyond a certain level due to interference can significantly impact client performance. We set LIR_{thresh} equal to 0.7 to identify those links that suffer more than 30% reduction in their LIR under interference and classify them as hidden terminals.

2) Rate anomaly: Rate anomaly is a well documented problem [23] in wireless environments. If a transmitter of a link operating at a high data rate (say 54 Mbps), carrier senses the transmitter of another link operating at a low rate (say 6Mbps), then the link operating at higher rate will experience significant slowdown in throughput (by a factor of 1/10 in this case). We classify a given link pair as a case of rate anomaly, when the ratio of their transmission rates is less than 0.2.

Both these issues are observed in both production networks. Table 5 shows the extent of hidden interference and rate anomaly in the two WLANs. The extent of hidden interference is rather limited (8% for WLAN1 and 11% for WLAN2). For comparison, Jigsaw [8] also reports that 5% of their links observe an LIR of less than 0.8. While limited on average, however, we do still observe, across both WLANs that hidden interference can lead to up to 70% LIR degradation for as many as 4% and 3% of the links in WLAN 1 and 2 respectively.

In terms of rate anomaly issues, we observe that for about 20% carrier sensing link pairs, the transmission rates differ by more than 80%. This could be one of the reasons for sudden performance slowdown experienced by perfectly good quality links in WLANs.

6 Conclusions

We presented a detailed evaluation of a passive, real time interference estimation mechanism (PIE). We showed that PIE is accurate in estimating link interference and can also adapt to changing interference patterns in real time. This enables PIE to be especially effective in realistic wireless environments, where client mobility, variable transmission rates, and bursty traffic result in changing interference scenarios, thereby limiting the usefulness of static bandwidth test mechanisms. Further, we showed that PIE is completely passive, does not require client support, and does not cause any network downtime, making it attractive for use in real WLAN settings. We have integrated PIE with interference mitigation mechanisms like centralized scheduling, transmit power control, and channel assignment and showed that PIE can enable these mechanisms to function efficiently and dynamically by providing an accurate conflict graph in real time. We also used PIE to monitor two production WLANs and demonstrated that PIE can diagnose certain performance issues in real systems.

Acknowledgments: We would like to thank the anonymous reviewers and our shepherd Hari Balakrishnan, whose comments helped bring the paper into its final form. V. Shrivastava, S. Rayanchu, and S. Banerjee have been supported in part by US NSF through awards CNS-1040648, CNS-0916955, CNS-0855201, CNS-0747177, and CNS-1059306.

References

[1] Madwifi atheros drivers. http://madwifi-project.org.
[2] D. Aguayo et al. Link-level measurements from an 802.11b mesh network. In *SIGCOMM*, 2004.
[3] N. Ahmed and S. Keshav. Smarta: A self-managing architecture for thin access points. In *CoNEXT*, 2006.
[4] N. Ahmed et al. Online estimation of RF interference. In *CoNext*, 2008.
[5] N. Ahmed et al. Measuring multi-parameter conflict graphs for 802.11 networks. In *MC2R*, 2009.
[6] K. Cai et al. Non-intrusive, dynamic interference detection for 802.11 networks. In *IMC*, 2009.
[7] R. Chandra, J. Padhye, A. Wolman, and B. Zill. A location-based management system for enterprise wireless LANs. In *NSDI*, 2007.
[8] Y.-C. Cheng et al. Jigsaw: solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM*, 2006.
[9] Y.-C. Cheng et al. Automating cross-layer diagnosis of enterprise wireless networks. In *SIGCOMM*, 2007.
[10] J. Eriksson, S. Agarwal, P. Bahl, and J. Padhye. Feasibility study of mesh networks for all-wireless offices. In *MobiSys*, 2006.
[11] J. Geier. Assigning 802.11b access point channels. *Wi-Fi Planet '04*.
[12] IEEE. Wireless LAN Medium Access Control (MAC) and physical layer (PHY) spec, IEEE 802.11 standard. *IEEE Standard 802.11*, 1999.
[13] R. Mahajan et al. Analyzing the MAC-level behavior of wireless networks in the wild. *SIGCOMM 2006*.
[14] A. Mishra et al. A client-driven approach for channel management in wireless LANs. In *INFOCOM*, 2006.
[15] D. Niculescu. Interference map for 802.11 networks. In *IMC*, 2007.
[16] J. Padhye et al. Estimation of link interference in static multi-hop wireless networks. In *IMC*, 2005.
[17] K. Ramachandran et al. Symphony: synchronous two-phase rate and power control in 802.11 WLANs. In *Mobisys*, 2008.
[18] M. Rodrig et al. CRAWDAD data set uw/sigcomm2004.
[19] M. Rodrig et al. Measurement-based characterization of 802.11 in a hotspot setting. In *SIGCOMM E-WIND*, 2005.
[20] E. Rozner, Y. Mehta, A. Akella, and L. Qiu. Traffic-aware channel assignment in enterprise wireless LANs. In *ICNP*, 2007.
[21] A. Sheth et al. MOJO: a distributed physical layer anomaly detection system for 802.11 WLANs. In *MobiSys*, 2006.
[22] V. Shrivastava et al. CENTAUR: realizing the full potential of centralized WLANs through a hybrid data path. In *MobiCom*, 2009.
[23] G. Tan and J. Guttag. Time-based fairness improves performance in multi-rate WLANs. In *USENIX*, 2004.
[24] M. Vutukuru, K. Jamieson, and H. Balakrishnan. Harnessing exposed terminals in wireless networks. In *NSDI*, 2008.
[25] White-paper from Meru Networks. Microcell deployments: Making a bad problem worse for pervasive wireless LAN deployments. http://www.merunetworks.com/pdf/whitepapers/.
[26] J. Yeo, M. Youssef, and A. Agrawala. A framework for wireless LAN monitoring and its applications. In *WiSe*, 2004.

SpecNet: Spectrum Sensing Sans Frontières

Anand Padmanabha Iyer, *Microsoft Research India*
Krishna Chintalapudi, *Microsoft Research India*
Vishnu Navda, *Microsoft Research India*
Ramachandran Ramjee, *Microsoft Research India*
Venkata N. Padmanabhan, *Microsoft Research India*
Chandra R. Murthy, *Indian Institute of Science*

Abstract

While the under-utilization of licensed spectrum based on measurement studies conducted in a few developed countries has spurred lots of interest in opportunistic spectrum access, there exists no infrastructure today for measuring real-time spectrum occupancy across vast geographical regions. In this paper, we present the design and implementation of SpecNet, a first-of-its-kind platform that allows spectrum analyzers around the world to be networked and efficiently used in a coordinated manner for spectrum measurement as well as implementation and evaluation of distributed sensing applications. We demonstrate the value of SpecNet through three applications: 1) remote spectrum measurement, 2) primary transmitter coverage estimation and 3) SpectrumCop, which quickly identifies and localizes transmitters in a frequency range and geographic region of interest.

1 Introduction

Radio Frequency (RF) spectrum measurement studies [9, 10, 5, 7] have confirmed that vast spans of licensed spectrum, deemed white-spaces, are heavily under-utilized. Such studies have helped make a case for allowing unlicensed devices to utilize unused parts of the spectrum opportunistically. Opportunistic Spectrum Access (OSA) is now increasingly seen as a necessity to meet the growing demands of wireless applications. In fact, the historic FCC ruling in 2008 permitting such opportunistic use (and in 2010 allowing use without the need to sense primaries) is a testament to the success of these measurement studies.

Nevertheless, most spectrum measurement studies to date have been conducted in a few developed nations, using only a handful of spectrum analyzers. Even today, the US remains the only country to have allowed an OSA model. Many more measurement studies, especially in developing nations, are perhaps necessary to make the OSA model accepted worldwide.

Further, these measurements represent static spectrum occupancy information over small parts of a country. While spectrum allocation is mostly static today, the adoption of OSA will result in much more dynamic use of spectrum. Thus, access to real-time spatio-temporal maps is beneficial for OSA devices to sense other OSA devices and determine which parts of the spectrum are free/lightly loaded. However, there exists no infrastructure today for measuring real-time spectrum occupancy across vast geographical regions.

Over the past few years, several researchers have proposed novel schemes for efficient media access and network design in white-spaces [3, 20]. Other researchers have proposed novel collaborative spectrum sensing techniques [11] to allow robust detection of spectrum occupancy. However, thorough evaluation of these techniques using real data is hard today. Further, cross-geographic questions such as "How do spatio-temporal access usage patterns in India differ from those in the US?" or "How would a certain OSA technique that works well in the US perform in the UK?" cannot be answered today.

The primary contribution of this paper is SpecNet—a platform that allows researchers across the world not only to conduct spectrum measurement studies remotely in real time, but also implement and test novel distributed collaborative spectrum sensing applications for OSA. SpecNet advances OSA in several ways. First, it helps gather spectrum data in many countries, thereby helping the adoption of the OSA model worldwide. Second, by providing real-time spectrum occupancy maps, OSA devices may be able to quickly identify lightly loaded parts of the spectrum. Third, it provides real trace data that can be used to evaluate novel research ideas in OSA. Finally, in countries such as India, where there is no readily available database of primary users, it can help create an accurate database that can be used by OSA devices.

In SpecNet (Section 4), participant owners of spectrum analyzers register and connect their instruments to the SpecNet server. Each owner volunteers to provide time periods when SpecNet users are allowed to use the instrument to remotely conduct experiments. SpecNet provides its users with a rich API implemented as XML-RPC calls. Thus, SpecNet users can develop and remotely execute measurements or distributed sensing applications in a programming/scripting language of their choice. To the best of our knowledge SpecNet is the first programmable distributed spectrum sensing platform of its kind. SpecNet can be accessed at [15].

SpecNet provides an API that supports three classes of users (Section 4.2). For sophisticated users, SpecNet provides full access to the low-level APIs of the spectrum analyzer. For policy users and others mainly interested in measurement data, say for longituidinal analysis, SpecNet provides APIs that allow access to historic measurement data that SpecNet collects and stores in a database. For other users such as network operators or government personnel, SpecNet provides a set of high-

level APIs that allow these users to write novel applications without having to worry about the intricacies of the spectrum analyzer. For example, a government user interested in spectrum occupancy data need only specify the part of the spectrum (*e.g.,* 500-800 MHz), the geographical boundary (*e.g.,* specified by a center and radius of a circular region), the time interval (*e.g.,* between 12:00 - 16:00 hrs today) and the minimum signal strength of the transmitter that needs to be detected (say -95 dBm). Behind the scenes, SpecNet determines the group of relevant spectrum analyzers and their respective settings that will help satisfy the measurement request, executes the task on these spectrum analyzers and delivers the results to the user. Other users such as OSA network operators may be interested in determining the coverage of their networks at locations where spectrum analyzers may not be available. SpecNet provides an interpolation tool that uses measurements from nearby spectrum analyzers to estimate power at the location(s) of interest.

Given that spectrum analyzers are expensive ($10-40K) and their time of availability for SpecNet's use might be restricted depending on the owner's needs, an important design goal for SpecNet is efficient management of spectrum analyzer time. When two or more spectrum analyzers lie in the region of interest, it may be possible to coordinate their measurements in a manner so as to reduce the overall scanning time while satisfying the user's request. One approach could be to partition the frequency spectrum equally among all the spectrum analyzers in the region of interest. Another approach is to leverage the spatial diversity in the locations of the spectrum analyzers and partition the scanning efforts geographically. Finally, a hybrid approach that combines these two approaches is also feasible.

Two fundamental tradeoffs underlying the very physics of spectrum measurements make this problem of partitioning the measurement task among spectrum analyzers a significant challenge. First, the *time-frequency uncertainty principle* dictates that the finer the resolution of the spectrum scan, the longer it takes to perform the scan. Second, weaker signals require longer scan times to be amenable to detection. Further, the heterogeneity in capability as well as processing speeds across different models of spectrum analyzers adds to the complexity. SpecNet considers these tradeoffs and uses a novel task partitioning scheme for scheduling individual spectrum analyzers (Section 5).

We demonstrate the power of SpecNet through three applications (Section 7). The first application is simply a spectrum scan that is performed across different countries, illustrating the ability to conduct remote measurements. The second application is a coverage estimation application that may be useful to network operators. The application first helps localize a TV transmission tower and then predict its footprint so that operators may avoid the primary owner of the spectrum. This is especially useful in developing countries where a database of primary transmitters is unavailable or incorrect. The third application is SpectrumCop, which may be of interest to government users. Today, it is hard to detect violators of spectrum policy unless a primary owner of the spectrum complains of interference. The SpectrumCop application allows a user to quickly detect and localize a transmitter in a given frequency range and geographic region, demonstrating the utility of SpecNet's coordinated sensing platform.

Thus, we make the following contributions:

- We present the design and implementation of a novel platform called SpecNet that allows spectrum analyzers around the world to be networked and used in a coordinated manner for remote measurement as well as testing and implementation of distributed sensing applications. SpecNet is open for access at [15].
- We present a scheduling algorithm for coordinating measurements among neighboring spectrum analyzers that optimizes spectrum analyzer usage time.
- Finally, we present three applications that demonstrate the value of the SpecNet platform.

2 Related Work

Measurement Studies. One of the earliest studies that aimed at quantifying spectrum usage [9] is by the Shared Spectrum Company. The study, conducted at six different locations in the US, concluded that the average occupancy of spectrum was about 5.2% in the 30 MHz to 3 GHz frequency range. A study by McHenry *et al.* [10] in Chicago and New York revealed that the occupancy was limited to 17% and 13% respectively. Since then, there has been a number of measurement studies [5, 7, 19] in different parts of the world. The common finding of all these studies has been that spectrum is heavily underutilized. In [4], authors derive various statistics from the collected data, and propose a prediction algorithm for channel availability.

All of these studies have been performed using a handful (maximum of 4 according to [4]) of spectrum analyzers scanning spectrum in a small geographical region in an uncoordinated fashion. In contrast, SpecNet provides a platform for coordinating spectrum analyzers across different geographical regions, thus opening doors to more interesting measurement studies. Further, it also enables building occupancy maps of large geographical areas over long durations for longitudinal analysis.

Whitespace Research. Whitespace networking has been gaining attention as an important research field in the networking community. In [3], the authors propose

a Wi-Fi like system built on UHF whitespaces. Yang *et al*. propose a distributed spectrum access technique using frequency agile radios transmitting in orthogonal frequencies [20]. Most of these proposals have been evaluated in restricted settings. We believe that SpecNet would aid whitespace research by allowing evaluation of proposals based on broader, more real-world data. For instance, spectrum measurement data from different continents could be used to evaluate detection techniques.

Cooperative Sensing & Sensor Networks. Cooperative sensing is a well explored topic [11, 16, 6]. The main focus of these papers is detecting a primary whose frequency of transmission and/or location is known. Moreover, the emphasis is on novel collaborative detection techniques. SpecNet and research in collaborative sensing are complementary to each other. For example, measurements from SpecNet can be useful for evaluating these collaborative detection algorithms while advanced collaborative detection techniques can be incorporated into the SpecNet platform as an API.

SpecNet uses Voronoi partitioning for optimizing scan time of spectrum analyzers. The use of Voronoi diagrams has been proposed in sensor networks as well [17, 2]. However, the main motivation for applying a partitioning scheme in sensor networks has been energy savings and/or interference avoidance. Thus, the problem formulations and objective functions are very different.

Testbeds/Platforms. A number of distributed research testbeds/platforms have been built by the community [12, 1, 18]. To the best of our knowledge, SpecNet is the first platform targeted at co-ordinating spectrum analyzers across geographical regions.

3 Spectrum Sensing Using Spectrum Analyzers - A Primer

In this section we attempt to answer the question,"what are the key settings and choices available to a spectrum analyzer user for spectrum scanning and how do they influence the spectrum sensing process?"

3.1 Spectrum Scanning - An Example

We begin with an example spectrum scan of an active wireless microphone depicted in Figure 1. When scanning using a spectrum analyzer, a user typically needs to specify two key parameters—the *scanning frequency range* and the *resolution bandwidth*. The frequency range, (f_{min}, f_{max}) in MHz, specifies that the user is interested in scanning the spectrum from f_{min} MHz to f_{max} MHz. In Figure 1, the scanning frequency range is $(702.05\ \text{MHz}, 702.35\ \text{MHz})$. Resolution bandwidth specifies the granularity in Hz at which the scan is to be performed—the lower the resolution bandwidth, the greater the observed detail in the scan.

Figure 1 depicts the results of the scan at four different resolution bandwidths. When the resolution bandwidth is 1 MHz, the microphone's transmission is not at all perceivable. Upon reducing the resolution bandwidth to 30 KHz, a single clear peak emerges indicating the microphone's transmission. Further reducing the resolution bandwidth to 10 KHz reveals even finer detail—three distinct peaks, which are the signature of an FM-modulated transmission. At 1 KHz resolution bandwidth, the three peaks are revealed as distinct sharp tones.

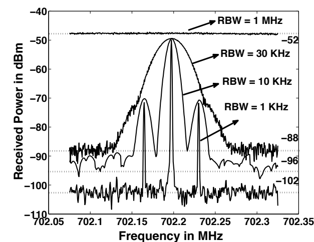


Figure 1: Effect of resolution bandwidth

As seen in Figure 1, a lower resolution bandwidth has two significant effects on the scan. First, greater detail about the signal structure is revealed and second, the noise floor is reduced (from -52 to -102 dBm).

3.2 Occupancy Detection

Often, the goal behind scanning the spectrum is *occupancy detection*, *i.e.,* to determine which parts of the spectrum have ongoing transmissions. Fundamentally, the problem of occupancy detection attempts to distinguish between signal and noise. While there are several varieties of occupancy detection schemes, perhaps the simplest scheme is to check whether the Signal to Noise Ratio (SNR) is greater than a certain threshold.

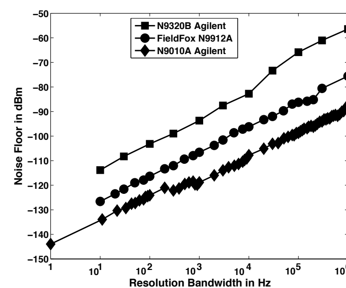


Figure 2: Noise floor versus resolution bandwidth

Dependence of noise floor on resolution bandwidth: As we saw in Figure 1, the noise floor depends on the resolution bandwidth of the scan. This decrease in noise floor arises from the fact that as frequency bins become finer, they accumulate less noise. *A lower noise floor typically results in a greater SNR and consequently more reliable occupancy detection.*

The noise floor (in watts) as a function of resolution bandwidth is typically given by

$$N \propto \rho \tag{1}$$

In Eqn 1, the proportionality constant depends on the spectrum analyzer model, the antenna, the cabling, *etc*. Figure 2 depicts the dependence of noise floor on resolution bandwidth for three different models of spectrum analyzer. While practical measurements indicate minor deviations in linearity, as seen in Figure 2, the linear model (Eqn 1) holds approximately true for all spectrum analyzers we used.

Dependence of detection range on resolution bandwidth: Typically, the farther a transmitter is from a spectrum analyzer, the lower the received power at the spectrum analyzer. The weaker the received signal, the lower the SNR and hence the less reliable its detection. *Detection range* of a spectrum analyzer at a certain resolution bandwidth is the farthest distance from which an ongoing transmission can be detected reliably.

Path loss models such as the Log Distance Path Loss (LDPL) model are typically used to estimate received power as a function of distance. The received power P_r at a distance d from a transmitter transmitting with power P_0 based on the LPDL model is given by

$$P_r = P_0 - 10\gamma \log(d) + L \tag{2}$$

In Eqn 2, γ (usually between 2 and 3 for outdoor environments) is the path loss exponent and L dB (usually modeled as a Gaussian with standard deviation between 5-10 dB for outdoor environments) is a random variable that captures variations in the signal due to fading effects.

If Δ is the minimum SNR required for reliable occupancy detection using a certain detection scheme, then in order to detect a transmission from a distance d, the noise floor must be Δ dB less than P_r, *i.e.,* $P_0 - 10\gamma \log(d) - \Delta$. Since noise floor is dictated by the resolution bandwidth (Eqn 1), this in turn implies that *one must choose a lower resolution bandwidth to reliably detect a transmitter that is farther away from the spectrum analyzer.* The dependence of detection range d on resolution bandwidth can be derived from (Eqn 1) (after converting from dB) as

$$\rho \propto \left(10^{\frac{P_0 - \Delta}{10}}\right) d^{-\gamma} \tag{3}$$

Eqn 3 indicates an important aspect of detecting transmissions from a distance, namely, *the maximum usable resolution bandwidth decreases super-linearly (as d^{γ}) with detection range*.

4 SpecNet Architecture

SpecNet is a shared infrastructure consisting of geo-distributed, networked, programmable spectrum analyzers that are contributed and used by the community. The following two goals drive the design of SpecNet. *1) Ease of Use:* We expect SpecNet to support the needs of three different classes of users. First, sophisticated users such as whitespace researchers will likely need real-time, low-level access to the full functionality of the spectrum analyzers. Second, some users such as spectrum policy researchers may simply need access to the data collected by the spectrum analyzers. Finally, users such as secondary network service providers or government personnel interested in spectrum monitoring may require high-level APIs that abstract the details/complexity of SpecNet and provide services such as tower localization or spectrum occupancy detection. *2) Efficiency:* Given that spectrum analyzers are expensive ($10-40K) and may be available to SpectNet for limited duration, it is important that the usage of spectrum analyzers be optimized where possible. Since the spectrum analyzers cannot be arbitrarily "time-sliced" for fine-grained sharing, optimization requires completing each task as efficiently as possible. We now present an overview of the SpecNet architecture.

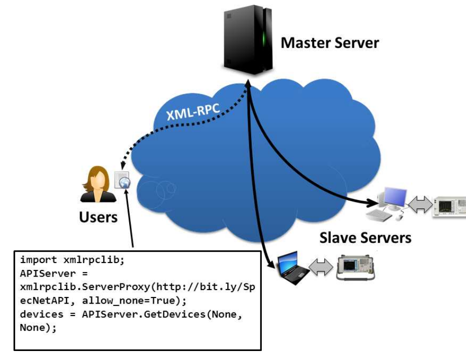


Figure 3: SpecNet Architecture

4.1 Overview

The SpecNet architecture is shown in Figure 3. It contains three key components: users or clients, slave servers that comprise laptops/PCs connected to spectrum analyzers, and master servers that manage the slave servers. The typical work-flow is as follows: clients submit jobs to the master servers; the master servers translate these jobs into spectrum analyzer commands based on Standard Commands for Programmable Instruments (SCPI) [14]. The master server also schedules these at the appropriate slave server nodes for execution at the desired/available time. The output of the commands is then either forwarded immediately to the client or the client is notified of when/where the output data from the submitted job would be available.

XML-RPC: In order to support a wide range of client platforms, the SpecNet service is exposed by the master servers as XML-RPC calls, *i.e.,* remote procedure calls that are encoded in XML and transported over HTTP using the XML-RPC standard. This allows clients to

post jobs using the SpecNet APIs from any Internet-connected node, written in any language of their choice.

Push-vs-Pull: The jobs posted to the master server can either be pushed to or pulled by the slave servers. While a pull-based publish-subscribe model is less complex in terms of state maintenance at the server, it is not suitable for SpecNet users who may want to execute jobs with inter-dependent API calls that require reaction at sub-second intervals (see the Spectrum Cop application in Section 7.3). We thus adopt a push-based model where a persistent TCP connection is maintained between the slave servers and the master servers and jobs are pushed to the slave servers.

Registration: Users contributing slave servers need to first register with the SpecNet master server. They may specify times during which the nodes are available to SpecNet. Upon completion of registration, a simple daemon is downloaded and executes on the slave server. This software establishes an outbound persistent TCP connection to the master server and another connection to the spectrum analyzer, thereby serving as a bridge between the master server and the spectrum analyzer.

Benchmarking: The master server first runs a suite of experiments to benchmark the fundamental characteristics such as noise floor and scan times of each spectrum analyzer (details in [8]). This benchmarking helps the master server efficiently schedule jobs at the slave server nodes. Further, this is also necessary for abstracting some of the low-level details of the spectrum analyzer through higher-level APIs, necessary for masking some of the heterogeneity among spectrum analyzers. We discuss this next.

4.2 APIs

As mentioned earlier, SpecNet is designed to support three classes of users. Table 4.2 lists a subset of the APIs supported by SpecNet.

For sophisticated users who require low-level access to the spectrum analyzer, SpecNet has a reservation API that users can use to reserve a block of time on the desired slave servers. The users can then issue their desired low-level commands, which are simply forwarded through the master server to the slave servers for execution.

For policy users and others who are interested mainly in spectrum usage data, possibly for longitudinal studies, SpecNet schedules up to 10% of the available time at each slave server for itself. During this time, the server performs a high resolution scan of the entire spectrum, stores this data in a SQL database and exposes this data to users through APIs such as `GetPowerSpectrumHistory()` or `GetOccupancyHistory()`. This stored data can also serve as a cache and may help respond (partly) to other submitted jobs.

The interesting challenges in SpecNet's design arise mainly in supporting the third class of users (*e.g.,* network operators). These users may require support for high-level APIs that abstract out many of the details of using spectrum analyzers. While we have designed a few of these APIs (6-9 in Table 4.2), we expect the set of high-level APIs to expand over time based on interest and through community contributions.

Localization and Interpolation: Estimating the geographical coverage of a primary transmitter is essential to creating a spectrum usage map. However, this requires knowledge of specifics of the transmitter such as its location and transmit power. Such information is usually not available or may be incorrect, especially in developing countries (Section 7.2).

In order to localize transmitters, SpecNet provides the `LocalizeTransmitter()` API that uses signal strength observed at spectrum analyzers from various locations but does not require input of parameters such as location and transmit power of the transmitter. Instead, SpecNet estimates these parameters that best explain the signal observations (in least mean square error terms) using well known path loss models such as Longley-Rice or Log Distance Path Loss (LDPL). The number of unknowns that can be estimated, however, fundamentally depends on the number of different locations from which signal strength was observed. In case of the LDPL model (Eqn 2), for example, if signal strengths from only three locations are available, SpecNet sets $\gamma = 3$, takes the transmit power (P_0) as input from the user and estimates the location through triangulation. If signal strength from four different locations are available, SpecNet can estimate P_0 and the transmitter location simultaneously by choosing $\gamma = 3$. When observations from five or more locations are available, SpecNet can estimate the transmitter location, transmit power P_0 and γ simultaneously that best fit the observations. Once the location of the transmitter and other parameters are determined, constructing a spectrum map is straightforward. SpecNet provides the `FindPowerAtLocation()` API that takes these parameters and predicts the likely received power at desired new locations (*e.g.,* locations with no spectrum analyzer).

Spectrum Occupancy Detection: The next two high-level APIs help users obtain spectrum occupancy at desired locations. The `GetPowerSpectrum()` is simply a spectrum scan over a given frequency range on a given device, except that users do not even need to specify the resolution bandwidth. Instead users can specify a region and desired minimum power level of transmitter to be detected. SpecNet then automatically chooses the best resolution bandwidth (based on the fundamental properties of occupancy detection discussed in Section 3)

#	API	Description
	Low-level APIs (*e.g.*, for sophisticated users)	
1	`GetDevices([Boundary], [Timespan])`	Returns a list of spectrum analyzer IDs. Fewer/no arguments possible.
2	`ReserveDevice(ID, Timespan)`	Reserves and returns success, if available.
3	`RunCommandOnDevice(ID,Command)`	Issues SCPI command to device and returns result.
	Commands to access stored data (*e.g.*, for policy users)	
4	`GetPowerSpectrumHistory(ID, Fs, Fe, Timespan)`	Returns (avg) power values from device for given time/frequency range (Fs-Fe).
5	`GetOccupancyHistory(ID/Boundary, Fs, Fe, Timespan, Threshold)`	Returns 0-1 list indicating occupancy in Fs-Fe at device or in region, based on threshold.
	High-level APIs (*e.g.*, for operators or government users)	
6	`LocalizeTransmitter(Boundary, Locations, Powers, Model, Parameters)`	Localizes transmitter inside area, given observed power level(s) at location(s) using Model (LDPL, HATA, Longley-Rice, etc.) .
7	`FindPowerAtLocation(Location, [Transmitter Parameters], Model, [Model Parameters])`	Interpolates power at new location given transmitter location/parameters and model; useful for estimating coverage of transmitter.
8	`GetPowerSpectrum(ID, Fs, Fe, [Boundary, P])`	Schedules a scan for given frequency range (SpecNet determines optimal resolution bandwidth) in order to detect minimum power level P in given area.
9	`GetOccupancy(ID/Boundary, Fs, Fe, P)`	Provides a 0-1 list corresponding to frequencies occupied at a device or region. P is the minimum transmitter power (SpecNet minimizes scan time).

Table 1: Core APIs supported by SpecNet

and returns the results. `GetOccupancy()` API goes further by allowing the user to specify a region of interest for detecting occupancy of signals above a given threshold, without even identifying the desired slave server IDs. This API is useful for applications like Spectrum Cop (Section 7.3), which monitor unauthorized spectrum usage. To support this API, SpecNet computes the optimal set of spectrum analzyers and their corresponding resolution bandwidth values that minimize scan time and returns the results. Optimizing scan time across multiple spectrum analyzers is a challenging problem which we discuss next.

5 Task Scheduling in SpecNet

SpecNet allows users to deploy and execute spectrum sensing applications in real time. Users expect their sensing tasks to be dispatched and completed as soon as possible. Consequently, SpecNet schedules participant spectrum analyzers in a manner so as to minimize task completion time. In this section we describe the challenges posed in the design of a task scheduler for SpecNet.

5.1 Scanning Time of a Spectrum Analyzer

For a spectrum analyzer, the time to perform a scan from f_{min} MHz to f_{max} MHz depends on two parameters namely, *span* $Q = f_{max} - f_{min}$ and the resolution bandwidth ρ used for the scan. Increasing the span requires a spectrum analyzer to scan a larger part of the spectrum and consequently requires a longer scan time. Scanning at a smaller resolution bandwidth requires a larger number of samples to be collected in order to reliably estimate the power in each of the finer frequency bins and hence, more time. For modern spectrum analyzers, the scan time may be modeled as

$$T \propto \frac{Q}{\rho} \quad (4)$$

In Eqn 4, T is the *scanning time*. The proportionality constant in Equation 4 can vary significantly across different models of spectrum analyzers as discussed next.

Theory versus Reality : Figure 4 depicts the scan times measured from different spectrum analyzers at different resolution bandwidths as a function of span. As seen from Figure 4, the dependence of scanning time on span Q is strictly linear as dictated by Eqn 4. Consequently, it is convenient to characterize scan times of spectrum analyzers in terms of *scan time per MHz*, τ. The scanning time for a scan from f_{min} to f_{max} is then determined by the product $(f_{max} - f_{min})\tau$.

Figure 5 depicts the measured scan times per MHz (τ) as a function of resolution bandwidth for three different models of spectrum analyzers in a log-log plot. Based on Eqn 4, the variation of scan times with resolution bandwidth should be linear. However, Figure 5 indicates *significant departure from linearity.* Rather the variation is *piece-wise linear*. For example, for FieldFox N9912A, the variation is linear in sections A-B and C-D separately. The piece-wise linearity arises because spectrum analyzers likely use different sets of circuits and modes for different ranges of resolution bandwidths and these circuits/modes presumably have different performance characteristics. To allow for these non-linearities, SpecNet maintains lookup tables $\tau(\rho)$ describing the scanning time per MHz for a given resolution bandwidth setting for each spectrum analyzer.

5.1.1 Minimizing Scan Time by Automatic Resolution Bandwidth Selection

When scanning a part of the spectrum, users often care about having a low noise floor. The noise floor, however, as discussed in Section 3, depends on the resolution bandwidth chosen. SpecNet allows users to request a scan by a remote spectrum analyzer by specifying the maximum tolerable noise floor. Behind the scenes, SpecNet determines the resolution bandwidth that provides for the fastest scan time that satisfies the required noise floor. In order to enable such an API, the SpecNet server maintains lookup tables that provide scanning times per MHz at various resolution bandwidths, for each Spectrum Analyzer connected to SpecNet.

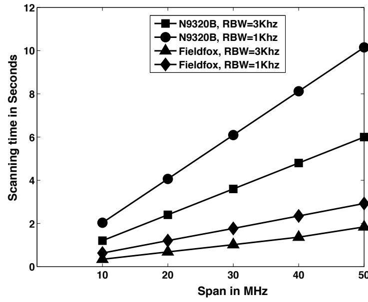


Figure 4: Scanning time versus span

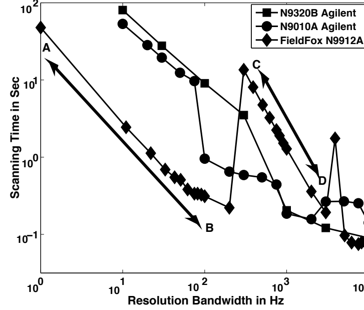


Figure 5: Scanning time per MHz versus resolution bandwidth

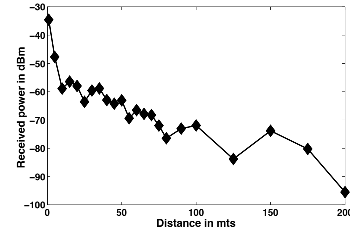


Figure 6: Decay in received signal strength for microphone

Dependence of Scanning Time on Detection Range: A greater detection range requires using a narrower resolution bandwidth (Section 3). This in turn implies that *to increase the detection range of a spectrum analyzer one must accept a longer scanning time*. More specifically, from Equations 3 and 4, scanning time depends on detection range as

$$T \propto \left(10^{-\frac{P_0-\Delta}{10}}\right) Qd^{\gamma} \tag{5}$$

Eqn 5 reveals a crucial aspect of sensing—namely, *scanning time increases super-linearly with increase in detection distance and linearly with span*. As described in Section 5.2, SpecNet uses this dependence to efficiently share load among spectrum analyzers given a scanning task.

To account for the deviations in scanning time from Equation 4 as depicted in Figure 5, given a detection range d , instead of using Eqn 5, SpecNet uses the lookup table $\tau(\rho)$ to determine the resolution bandwidth that has the fastest scanning time per MHz while ensuring a minimum noise floor of $P_0 - 10\gamma \log(d) - \Delta$. $P_0 = -50$ and $\Delta = 10$ are chosen as default unless specified by the user and $\gamma = 3$ is chosen as a conservative estimate.

Evaluation: Given a detection range, SpecNet chooses a resolution bandwidth so as to minimize scanning time. How well does the resolution bandwidth selection scheme work in practical deployments? A resolution bandwidth chosen too low will take too long to scan while a resolution bandwidth chosen too high will not provide the necessary SNR to allow detection. There are several practical considerations. First, the path loss exponent is not a fixed quantity and depends on the nature of the environment. Line of sight and non line of sight paths offer different path loss characteristics. Further, significant signal attenuation often occurs due to walls in indoor environments.

To answer this question, we tested SpecNet in a real deployment at the Indian Institute of Science (IISC) campus as depicted in Figure 7 on two different models of spectrum analyzer. The campus is lush with very dense trees and this provided an excellent opportunity to evaluate SpecNet in various scenarios such as Line of Sight (LOS), Non-Line of Sight (N-LOS) and Indoors. In Figure 7, two different models of spectrum analyzer are located at O, while a wireless microphone was placed at six different locations, two each in the LOS, NLOS and indoor categories. In each of the six detection experiments, the detection range was set to the exact distance between the microphone and the spectrum analyzer. P_0 was set to -35 dBm which was determined by measuring the power of microphone at a distance of 1m. For all our experiments we fixed $\Delta = 10$ dB. In other words, given a detection range, SpecNet must choose the resolution bandwidth that provides the minimum scanning time while ensuring that the SNR is a minimum of 10 dB. Table 8 provides a summary of the results.

Line of Sight: As seen from Table 8, for both the LOS experiments and for both spectrum analyzers, SpecNet chose a very conservative noise floor—while the target SNR is 10 dB, the observed SNR is about 25 dB. Figure 6 depicts the decay of signal strength with distance for the microphone in line of sight. The path loss decay exponent γ was estimated to be around 2.5, however, SpecNet conservatively chooses $\gamma = 3.0$ in estimating the target noise floor. This results in the conservative choice of the resolution bandwidth.

Non Line of Sight: For *NLOS experiments*, the resolution bandwidth choice of SpecNet allows for an SNR close to the target 10dB for both spectrum analyzers indicating that γ was closer to 3 for these experiments.

Indoors: When the microphone was kept *indoors*, however, SpecNet finds itself underestimating the signal decay. For example, in both the experiments, the chosen resolution bandwidths allow only SNR of about 6 dB rather than 10 dB.

While choosing a conservative resolution bandwidth ensures detection, it results in longer scanning times. What is the loss in scanning time due to the conservative choices of resolution bandwidth? To answer this question, we attempted to detect the microphone at sev-

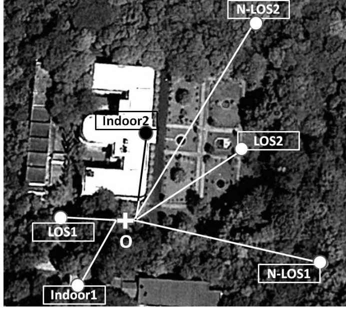


Figure 7: Occupancy detection using a single spectrum analyzer

	Distance in mts	SNR in dB		Loss in Sec Scanning Time	
Line Of Sight	31	24	28	0.005	0.018
	71	25	28	0.016	0.046
Non-Line Of Sight	124	15	23	0.123	2.23
	131	17	24	0.123	2.24
Indoor Locations	35	16	6	0.005	0.0
	50	0.2	8	0.0	0.0

Figure 8: Performance of Resolution Bandwidth Selection in SpecNet; the two columns for SNR and Scanning time represent two different spectrum analyzers

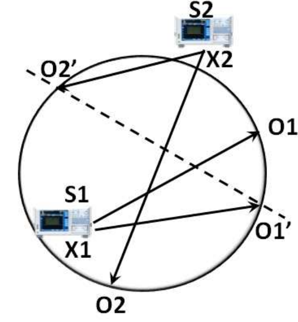


Figure 9: Occupancy detection using two spectrum analyzers

eral different resolution bandwidths without the use of SpecNet's resolution bandwidth selection. We then determined the optimal resolution bandwidth for each experiment that allowed an SNR of 10 dB. Table in figure 8 depicts the loss in scanning time in seconds due to the sometimes conservative choice of SpecNet for each experiment. As seen from table, the loss in scanning time is in the range of a few milliseconds most of the time and up to a few seconds in some cases. Thus, we conclude that the automatic resolution bandwidth estimation in SpecNet works as intended.

5.2 Occupancy Detection

In many practical applications of occupancy detection, users are interested in spectrum occupancy in a specific geographic region. For example, "are there any ongoing transmissions in the spectrum range 700 MHz to 800 MHz within a 5 km radius of my location?" SpecNet allows users to specify a circular region specified by a center and a radius for spectrum measurement. Behind the scenes, SpecNet determines the set of relevant spectrum analyzers that can be used to accomplish this task. Any spectrum analyzer whose maximum detection range (determined by the lowest resolution bandwidth) overlaps with the user-specified region of interest is deemed relevant. When there are multiple relevant spectrum analyzers, SpecNet schedules the scanning task load among them so as to minimize the overall scanning time.

5.2.1 Load sharing across multiple spectrum analyzers

There are two distinct dimensions along which a scanning task can be shared among multiple spectrum analyzers, namely, spectrum and geography. *Spectral load sharing* involves different spectrum analyzers scanning complementary parts of the spectrum while *geographical load sharing* involves different spectrum analyzers scanning different spatial sections of the overall geographical area of interest. SpecNet uses a combination of both these techniques to minimize overall scanning time.

The Scheduling Metric: If n different spectrum analyzers are scheduled to share a certain task load, they scan in parallel and accomplish their respective sub-tasks in parallel. Suppose that the ith spectrum analyzer takes time T_i to complete its assigned sub-task. The task is deemed complete when all spectrum analyzers have accomplished their respective sub-tasks. Since all spectrum analyzers are tasked in parallel, the time to task completion is given by $T = \max(T_1, T_2, \cdots, T_n)$. The goal of the SpecNet task scheduler is to minimize the task completion time. Hence, SpecNet attempts to schedule various spectrum analyzers in such a manner that the maximum over all sub-task completion tasks is minimized *i.e.,* in a *min-max* manner.

Spectral Load Sharing: Figure 9 depicts a circular region of interest and two spectrum analyzers S1 and S2 located at X1 and X2 that can potentially be used to scan the circular region of interest. Suppose that the user needs to scan from f_{min} MHz to f_{max} MHz. S1 and S2 could then share the task such that S1 scans from f_{min} MHz to $f_{min}+Q_1$ MHz, while S2 scans from $f_{min}+Q_1$ MHz to f_{max}. Such spectral load sharing results in a reduction in span for the participant spectrum analyzers, thus reducing the overall scanning time.

In the above example Q_1 must be chosen in a manner so that the maximum of the scanning times of S1 and S2 are minimized. In order to detect any transmission in the entire region of interest, S1 must have a detection range equal to $|\overline{X_1O_1}| = d_1$ where O1 corresponds to the farthest possible transmitter location within the region of interest from S1 (as depicted in Figure 9). Similarly, the detection range of S2 should be $|\overline{X_2O_2}| = d_2$ in order to detect any transmitter in the region of interest. Let τ_i be the minimum scanning time per MHz for spectrum analyzer S_i required to achieve a detection range of d_i. Then the overall scanning time is given by $\max(\tau_1 Q_1, \tau_2 Q_2)$, where $Q_2 = f_{max} - f_{min} - Q_1$. The optimal choice then corresponds to when

$$Q_1 : Q_2 = \frac{1}{\tau_1} : \frac{1}{\tau_2} \tag{6}$$

Eqn 6 can be easily generalized to spectral partitioning for several spectrum analyzers. In case of several spectrum analyzers, *the span of spectrum allocated to each spectrum analyzer is inversely proportional to the minimum scanning time per MHz required to scan the circular region of interest.*

Geographical Load Sharing: Another way to share the load between S1 and S2 (Figure 9) is to partition the region of interest geographically by requiring them to scan only parts of the region of interest rather than the entire region. In Figure 9, the region is divided into two sections by the line $|\overline{O_1'O_2'}|$. S1 and S2 are deemed responsible to scan each of the two sections. The advantage of partitioning in this manner is that individual spectrum analyzers can now use a smaller detection range. As seen in Figure 9, S1 and S2 use detection ranges equal to $|\overline{X_1O_1'}| = d_1' < d_1$ and $|\overline{X_2O_2'}| = d_2' < d_2$ respectively. As described in Equation 5, reduced detection range implies reduced scanning time. Thus, each of the spectrum analyzers takes a shorter time to scan its respective region—thus reducing overall task completion time.

Since every spectrum analyzer scans a different geographical region, each must scan the entire spectrum of interest f_{min} to f_{max}. If the scanning times per MHz of n geographically task sharing spectrum analyzers are given by τ_1,τ_2,$\cdots$,τ_n, then the over all task completion time will be $\max(Q\tau_1, Q\tau_2, \cdots, Q\tau_n)$. Consequently, in order to minimize over all task completion time, we need $\tau_i = \tau, \forall i$ such that τ is minimized while ensuring that the entire area of interest is covered.

First consider the case of homogeneous spectrum analyzers. Ensuring equal τ_i translates to ensuring equal maximum detection ranges to all the spectrum analyzers. This problem can be optimally solved using Voronoi partitioning with each spectrum analyzer being treated as a Voronoi site. Each Voronoi cell, then, would correspond to the geographical region assigned to the spectrum analyzer. The resolution bandwidth of each spectrum analyzer would correspond to the detection range required to accommodate the farthest point in its Voronoi cell.

Now consider the case of heterogeneous spectrum analyzers. Since the scanning times of different analzyers are different, standard Voronoi partitioning is no longer optimal. Instead, the SpecNet scheduler performs a modified version of Voronoi partitioning – equal detection time partitioning – where proximity is measured in terms of detection time rather than Euclidean distance.

Given the non-linear and discontinuous nature of dependence of detection time on detection range (Equation 5), to the best of our knowledge there exists no known exact solution to this partitioning problem. Consequently we resort to solving the problem numerically. The entire area of interest is sampled at several locations generated randomly over the area of interest. Each random location is then assigned to its nearest spectrum analyzer in terms of the scan-time required to detect a transmitter at that grid point. Note that if a point is located beyond the detection range of a spectrum analyzer, the corresponding scanning time is set to infinity. Finally, each spectrum analyzer is assigned a resolution bandwidth by setting its detection range to the farthest random location assigned to it. The run-time complexity of this numerical scheme depends on the number of random points chosen. In our implementation we generated random locations with a density of 1 location per sq meter. For an area of 1 Sq Km (1×10^6 random locations) we found that geographic partitioning took under a few hundred milliseconds on the SpecNet server.

5.2.2 Geographical versus Spectral Load Sharing

Which of the above two load-sharing schemes should we use and under what circumstances? To answer this question we describe the results of two experiments conducted in the Indian Institute of Science (IISC) campus, depicted in Figures 10a and 10b, scanning from 700-800 MHz. In each of the experiments we compared three different scheduling methods. In *Best Select,* the spectrum analyzer that can accomplish the task in the shortest time is selected and used to accomplish the scanning task without any load sharing. We compared Best Select with spectral and geographical load sharing.

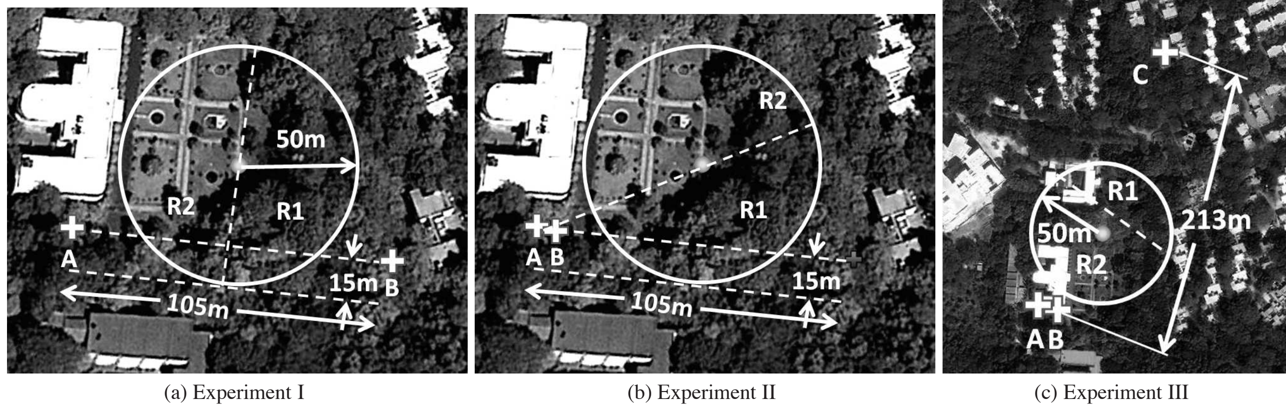
Experiment I : Two identical spectrum analyzers (both N9320B Agilent models) were placed 103 m apart at A and B as depicted in Figure 10a. The region of interest was specified as a circle of radius 50 m.

Experiment II : Two identical spectrum analyzers (both N9320B Agilent models) were both placed at location A and the region of interest was specified as a circle of radius 50 m as shown in Figure 10b.

Experiment	Best Select in sec	Spectral in sec	Geographical in sec
Experiment I	1054	561	129
Experiment II	1054	561	1054

Table 2: Comparison of load sharing schemes

Results of Experiment I : As depicted in Table 2, since the spectrum analyzers are identical, the optimal spectral load sharing resulted in both the spectrum analyzers taking an almost equal amount of time (in practice a slight difference in their noise floors resulted in one spectrum analyzer scanning a bit more spectrum than the other). Consequently, spectral partitioning completed about twice as fast as Best Select. Curiously, *geographical load sharing completed almost five times faster than spectral load sharing*. In this particular experiment, Voronoi partitioning resulted in two halves of the circle indicated by regions R1 and R2 in Figure 10a. Conse-



(a) Experiment I (b) Experiment II (c) Experiment III

Figure 10: Comparison of scheduling schemes

quently, the detection range required for each of the spectrum analyzers in geographical load sharing was smaller than that required in spectral load sharing. Eqn 5 reveals that scanning time decreases super-linearly as detection range, explaining the 5x gains.

Results from Experiment II : As depicted in Table 2, since the spectrum analyzers are co-located and identical, optimal spectral load sharing assigns two halves of the span to each spectrum analyzer. Consequently, spectral load sharing performs approximately twice as well as scheduling without load sharing. Here, however, geographical load sharing performs exactly the same as having no load sharing and takes twice as long as spectral partitioning! The Voronoi partition for the experiment is indicated by the dashed line separating R1 and R2 in Figure 10b. The maximum detection range required by each of the two spectrum analyzers to cover their respective partitions is actually almost the same as that required to cover the entire circular region of interest. Since both the spectrum analyzers scan the entire spectrum, one of the spectrum analyzers is actually redundant. *This experiment shows that when spectrum analyzers are very closely located, spectral partitioning can be more advantageous than geographical partitioning*.

5.2.3 Geo-Spectral Load Sharing

Spectral and geographical task sharing, as described in Section 5.2.1, each optimize along a single dimension only, namely either frequency (spectral) or area (geographical). As seen from Experiments I and II (Section 5.2.2), while geographical task sharing may be superior to spectral in some scenarios, the opposite may be true in others. A more general task partitioning scheme then is *geo-spectral* partitioning—where optimization is performed simultaneously along both the spectral and geographic dimensions.

Optimal geo-spectral task sharing, where spectrum analyzers are assigned a combination of frequency range and geographical area to minimize overall task completion time while ensuring that the entire area and spectrum of interest are covered, falls under a class of non-convex optimization problems for which, to the best of our knowledge, there exists no known exact solution. However, Experiments I and II (Section 5.2.2) reveal two key observations that allow us to develop a heuristic to enable geo-spectral task sharing. First, geographical partitioning typically out-performs spectral partitioning owing to the super-linear relationship between detection range and scanning time. Second, when spectrum analyzers are located near each other, spectral partitioning tends to outperform geographical partitioning.

In order to facilitate explanation of our heuristic for geo-spectral task sharing, we introduce the notion of a spectrally sharing cluster (SSC) of spectrum analyzers – a set of spectrum analyzers that share their scanning tasks spectrally over the same geographical region (possibly over only a small part of the entire region of interest). An SSC can be replaced by a single representative Virtual Spectrum Analyzer (VSA). The distance of a location from this VSA is then the maximum over the distances all spectrum analyzers in the corresponding SSC, since even the farthest constituent spectrum analyzer must detect occupancy at this location. The occupancy detection time for any location using the VSA is determined by optimally partitioning the spectrum among the constituent spectrum analyzers in the corresponding SSC (as described in Section 5.2.1). The union of two SSCs yields a VSA comprising the union of all constituent spectrum analyzers in both SSCs.

Our geo-spectral task sharing heuristic for n spectrum analyzers is initialized by creating n SSCs, each comprising a single distinct spectrum analyzer and performing geographical task sharing on them. The algorithm is a greedy iterative scheme, where at each step, pairwise SSC unions are considered in order to determine if overall task completion time can be reduced. In order

Model	Frequency Range	RBW steps
Agilent N9320B	9 KHz- 3 GHz	11 (10 Hz - 1 MHz)
Agilent Fieldfox N9912A	5 KHz - 6 GHz	36 (10 Hz - 1 MHz)
Agilent EXA N9010A	9 KHz - 26.5 GHz	62 (1 Hz - 8 MHz)
Agilent PSA E4440A	3 Hz - 26.5 GHz	68 (1 Hz - 8 MHz)
Hewlett-Packard E4403B	9 KHz- 3 GHz	15 (10 Hz - 5 MHz)

Table 3: Spectrum analyzer models used in SpecNet

to determine overall task completion time given a set of SSCs, each SSC is replaced by its corresponding VSA and geographical partitioning is performed on this set of VSAs. The SSC pair union that results in the maximum reduction in overall task completion time is accepted for the next iterative step. The procedure continues until no further opportunities to unite SSCs exist that can reduce the overall task completion time. In the worst case, the algorithm terminates in n steps, as at each step the number of SSCs decreases by 1. As, at each step all pairs of SSCs must be explored, the worst-case running time of this algorithm is $O(n^3)$. Since spectral sharing typically yields benefits only when two spectrum analyzers are "close", in practice the running time can be reduced to $O(n^2)$ by considering a fixed number of closest SSCs rather than all possible SSC pairs at each step.

Figure 10c depicts an example of Geo-Spectral load sharing. The scanning frequency range was chosen as 700 MHz to 800 MHz. Spectrum analyzers S1, S2 and S3 are located at A, B and C respectively. S3 (Fieldfox) is a much faster spectrum analyzer compared to S1 and S2 (both N9320B Agilent). The circular region of interest is geographically partitioned into two regions R1 and R2. S1 and S2 scan region R1 using spectral load sharing while S3 scans the entire spectrum in geographic region R2. To compare the performance of geo-spectral partitioning we also tried scheduling using the purely geographic and spectral schemes. Geographic load sharing took 1205 seconds; spectral load sharing 1118 seconds; and geo-spectral load sharing only 526 seconds.

In summary, load sharing across multiple spectrum analyzers is a challenging problem. SpecNet's Geo-Spectral load sharing algorithm is able to achieve 2-5X speedup compared to using a single spectrum analyzer in our experiments.

6 Implementation

The SpecNet platform is accessible at [15] via a web service API. It consists of a *master server* that manages several *slave servers*.

6.1 Master Server

The *master server* performs two major functions—first, it exposes an API (Section 4) which the SpecNet clients/users utilize to write programs and second, it manages all the *slave servers* connected to it.

As mentioned in Section 4, the API is exposed as XML-RPC calls to allow access from a wide-range of platforms. The *master server* implements a push-based model and thus, TCP connections to the slave servers are kept persistent using heartbeats. The current implementation of the master server is centralized and consists of approximately 5000 lines of C# code. However, partitioning of the slave servers along geographic boundaries is possible, thus allowing distributed execution across multiple master servers if scalability concerns arise.

One of the key challenges in managing slave servers is dealing with the heterogeneity of spectrum analyzers. As shown in Table 3, spectrum analyzers differ in their supported resolution bandwidth steps and frequency range of operation. Further, as discussed earlier, scan times (Figure 5) and noise floor (Figure 2) also vary across spectrum analyzers. SpecNet accounts for each of the above variations through a novel, automatic remote benchmarking process, described in detail in [8], that allows the master server to quickly build up a lookup table of scan times and noise floor values at different resolution bandwidth steps for each of its slave servers.

6.2 Slave Servers

The *slave server* is a small piece of software that runs on a desktop or laptop that are directly connected to the spectrum analyzer. The main task of the *slave server* is to act as a bridge between the spectrum analyzer connected to it and the *master server*. To avoid issues with NAT/-firewalls, the *slave server* initiates an outbound TCP connection on port 22 to the *master server*. It also connects to the local spectrum analyzer through VISA. Once connected, it translates commands from the master server to the spectrum-analyzer-specific-commands, runs spectrum scans, and returns the results.

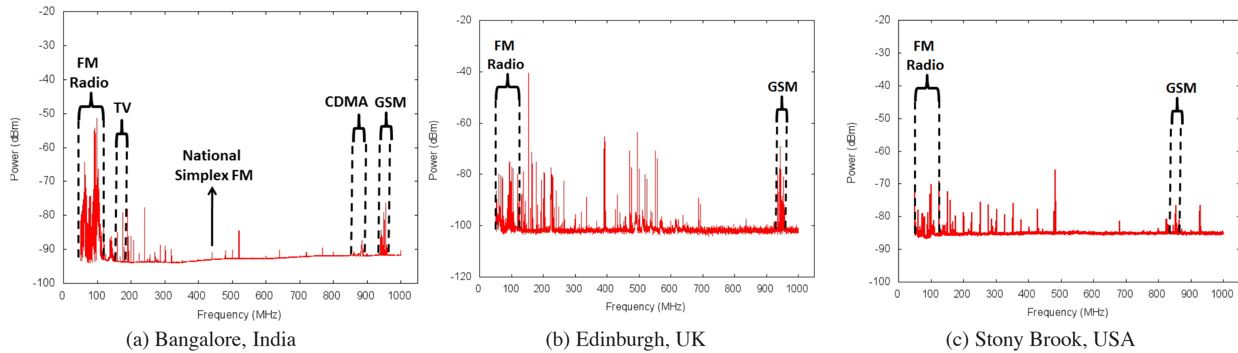
In order to support multiple platforms, we have implemented the slave server in Python in approximately 1000 lines of code. We use the PyInstaller [13] package to generate platform specific (Windows & Linux as of today) executables.

7 Applications

In this section, we present three example user applications on the SpecNet platform that highlight the simplicity of building a networked, geo-distributed system of spectrum analyzers.

7.1 Remote Spectrum Measurement

In this section we demonstrate how SpecNet can be used to make spectrum measurements anywhere in the world. The user code fragment written in Python is shown in Listing 1. One simply needs to connect to the SpecNet server, identify available devices in the region of interest and then use the `GetPowerSpectrum()` API to obtain power values in the desired parts of the spectrum. This data can be used, for example, to compare available free spectrum in different parts of the world or as



(a) Bangalore, India (b) Edinburgh, UK (c) Stony Brook, USA

Figure 11: Spectrum occupancy in various geographic regions

traces for evaluation of new white-space protocols such as WhiteFI [3].

Listing 1: Code snippet for remote measurement.

```
# connect to SpecNet server
apiServer = xmlrpclib.ServerProxy(
  "http://bit.ly/SpecNetAPI",
  allow_none=True);

# Find devices from region of interest
devices = APIServer.GetDevices(
  [55.944350, -3.187745, 500.0], None);
for device in devices:
    power_vals = APIServer.GetPowerSpectrum(
     device['ID'], Fs, Fe, 1e3);
```

At the time of writing, in addition to a few spectrum analyzers in Bangalore (India), we had one spectrum analyzer in Stony Brook (USA) and one in Edinburgh (UK) that were connected to SpecNet. Figure 11 shows the spectrum measurements at these three sites located in three different continents, demonstrating the world-wide reach of the SpecNet platform. As seen from Figure 11, spectrum measurements at each of these locations across the world clearly identify the well-known transmitters such as FM, TV, *etc*., and the available spectrum whitespaces.

7.2 Primary Coverage

The next example application determines the spatial footprint of a TV transmitter located within a large city. This may be useful for whitespace network operators in planning their deployments. Determining the footprint of a TV transmitter invariably requires knowledge of its location. While accurate databases of these locations are available in countries such as the US, such a database is not readily available in many developing countries, including India. We tried to obtain this information by contacting the Indian government agencies via postal mail (under the Right-to-Information Act). While we received information on about 150 TV tower locations (out of an estimated 700 towers), we found many inaccuracies in the data. For example, one tower's location was mapped well into a bay! Upon analyzing this TV tower data for five cities (ground truth based on Wikimapia), we found localization errors to range between 2-83 km (average 22 km, median 5 km). We now highlight how SpecNet could be used as a low-cost solution to improve the coverage and accuracy of the existing TV tower database.

Listing 2: Code snippet for primary coverage.

```
# Get Spectrum Analyzers in region
area_of_interest = [13.02236,77.56558, 100000.0];
devices = APIServer.GetDevices(area_of_interest, None);

# Get Power Spectrum Values
for device in devices:
    power_vals = APIServer.GetPowerSpectrum(
        device['ID'], Fs, Fe, 1e3);
    power_vals.append(average(power_vals));
    observation_locations.append([device['latitude'],
            device['longitude']]);
# Localize
if number_of_locations < 5
    localization_res = APIServer.LocalizeTransmitter(
     area_of_interest, observation_locations,
     power_values, 'LDPL', [-35.0, 3.0]);
else
    localization_res = APIServer.LocalizeTransmitter(
     area_of_interest, observation_locations,
     power_values, 'LDPL', None);

# Interpolate
pow = APIServer.FindPowerAtLocation(new_location,
 [localization_res], 'LDPL', None);
```

The code snippet for this application is shown in Listing 2. The region of interest is identified and power spectrum values from devices in that region are obtained. Then the TV transmitter is localized using the `LocalizeTransmitter()` API. Finally, a path loss model is used to build the spatial footprint of the TV transmitter. The API `FindPowerAtLocation()` is then used to determine the received power at desired new locations.

Bangalore city has one terrestrial TV transmitter. For the purpose of evaluation in a large-scale setting, we needed data from multiple spectrum analyzers at different locations in the city. Also, the accuracy of the localization API depends on the number of measurement locations. However, at the time of evaluation we only had access to four slave servers inside Bangalore. To get

around this problem, we modified the master server to allow mobile slave servers to connect to it. This enabled us to gather data from multiple locations in the city using just one mobile slave server by driving on the major roads and highways of the city. Figure 12 depicts the locations in the city where measurements were collected.

Figure 13 shows the TV tower localization error mean, 25th and 75th percentile (y-axis) as the number of measurement locations are varied (x-axis). To generate each point in Figure 13, twenty subsets of locations were randomly picked from the set of all measurement locations. We see that even when the number of measurement locations is between 5-10, the mean localization error varies between 2.5-3.8 km. This demonstrates that even by using measurements from a small number of spectrum analyzers in each city, the gaps and inaccuracies in the government database can be corrected significantly.[1] As the number of measurement locations is increased to 100, we see that the localization error goes below 0.5 km. While it is unrealistic to assume that SpecNet would have over 100 spectrum analyzers in each city, an alternative is to have spectrum analzyers that are mobile as part of SpecNet—we plan to look into this in the future.

Figure 14 shows the mean, 25th and 75th percentile errors in signal strength predictions obtained by using the interpolation API. The mean signal error varies between 6 to 8 dB, similar in magnitude to the expected signal variations due to the environment.[2] Thus, using SpecNet to calculate coverage of a primary transmitter can provide a good estimate to an operator.

7.3 SpectrumCop

Our final application demonstrates the two key features of SpecNet: 1) simplicity of writing a complex real-time application through the use of high-level APIs and 2) efficiency of SpecNet in scanning a wide frequency range when more than one spectrum analyzer is available, in order to detect violators quickly.

The goal of this application is to quickly detect a static narrow-band transmitter within a certain geographical region of interest and then localize the transmitter. The transmitter can be operating anywhere within a wide frequency range. This application is especially useful for, say, government officials to monitor unauthorized transmitters in a certain band.

The code snippet for this application is shown in Listing 3. The application uses the `GetOccupancy()` API for the transmitter detection part, which basically tasks one or more spectrum analyzers in the vicinity to perform scans at an appropriate resolution bandwidth and frequency range. The result of this API call is an occupancy list, which indicates frequencies that have ongoing transmissions. A more detailed spectrum measurement is then performed only in the region around the detected frequency. The results of the scan are then fed to the `LocalizeTransmitter()` API to determine the location of the transmitter.

Listing 3: Code snippet for SpectrumCop.

```
# Find occupancy in desired region
bound = [lat, lng, radius];
options = [lat, lng, radius, min_power_to_detect];
occupancy_list = APIServer.GetOccupancy(bound,
 start_frequency, end_frequency, min_power_detect);

# Get power spectrum for transmitter frequency
for occupancy in occupancy_list:
    if (occupancy['Occupied'] == 1):
        new_f_start = occupancy['Frequency'] - 250e3;
        new_f_end = occupancy['Frequency'] + 250e3;
        devices = APIServer.GetDevices(bound, None);
        for device in devices:
            locs.append([device['Latitude'],
                         device['Longitude']]);
            results[device['ID']] = APIServer.
             GetPowerSpectrum(device['ID'],
             new_f_start, new_f_end,
             options); # Actual call in new thread.
        break;

# Localize transmitter based on power measurements
for r in results:
        powers.append(max(r));
print APIServer.LocalizeTransmitter(bounds, locs,
 powers, 'LDPL', [P, 3.0]);
```

Evaluation: We used this application to detect and localize a microphone in a region of 75 meters radius in IISc. The setup consisted of 3 spectrum analyzers that were placed near 3 corners of the region of interest. The microphone transmits in a 250 KHz narrow band and the frequency range of the search space is set to 3 MHz. The SpectrumCop application detected the microphone perfectly and localized it to within 20 meters of the actual location. The entire process of detecting and locating the microphone took 165 seconds.

8 Limitations

First, spectrum analyzers are expensive equipment that researchers have procured for specific needs. It may not be easy to convince owners to volunteer this resource to the community, especially during the bootstrapping stage where the benefit of the platform is not clear to the owner. To date, we have approached a few of our acquaintances and have observed mixed results. In the long run, perhaps governments may be willing to sponsor a set of spectrum analyzers dedicated for SpecNet use.

Second, spectrum analyzers are typically used inside labs that may be in basements or deep inside buildings. Our measurements indicate that buildings can add 5-20 dB of attenuation (20dB in the basement for FM/TV

[1] Note that we used basic triangulation to locate the T.V tower, it may be possible to achieve a higher accuracy through more sophisticated localization schemes proposed in literature.

[2] In our implementation we used a simple log distance path loss model. The use of more sophisticated path loss models such as those that use terrain information may provide more accurate predictions

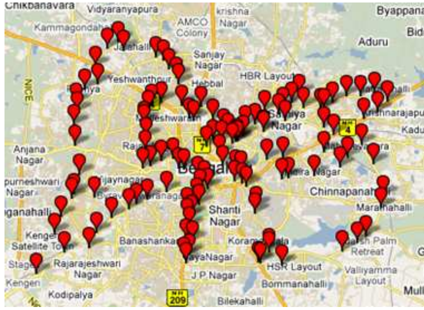


Figure 12: Measurement locations

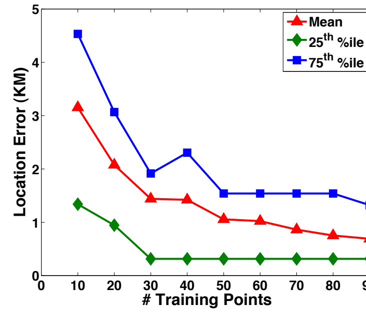


Figure 13: TV Tower Localization

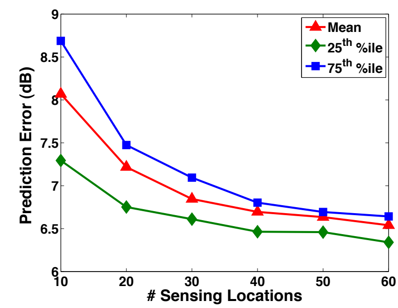


Figure 14: Interpolation results

transmissions) which restricts the detection range of the analyzer. If the owner can be convinced to mount the antenna near a window, the utility of the spectrum analyzer can be significantly increased. To minimize variability due to antenna placements, SpecNet can choose to only include spectrum analyzers with unobstructed antennas.

Finally, we have not considered the privacy/security implications of allowing remote scanning of the spectrum. For now, SpecNet only exposes the power values measured from the spectrum scan. Thus, it prevents direct security and privacy threats such as fine-grained traffic monitoring or user tracking. Advanced spectrum analyzers can provide time domain (I/Q) samples of the scan and support for these features in SpecNet would require sophisticated controls for privacy and security.

9 Conclusion

After the FCC ruling in the U.S. allowing opportunistic access to portions of licensed frequency bands, there has been tremendous interest in both academia and industry in developing novel wireless techniques and products that take advantage of the new rules. A key requirement for enabling this new ecosystem is a measurement infrastructure that can provide real data. SpecNet fulfills this critical need by enabling geographically distributed spectrum analyzers to be networked, thereby allowing both real-time remote measurements as well as collection of historic spectrum usage data. Furthermore, SpecNet exposes an API that allows users to build interesting distributed sensing applications like SpectrumCop with relative ease. There is still a lot of work left to achieve our goal of building a planet-scale networked spectrum analyzer testbed, but we believe SpecNet provides a good base to build upon.

10 Acknowledgements

We thank our shepherd, Brad Karp, and the anonymous reviewers for their constructive comments. We also thank Arsham Farshad, Mahesh Marina, and Akshay Athalye for helping us conduct remote spectrum measurements.

References

[1] http://www.emulab.net/.

[2] Ammari, H., and Das, S. Promoting heterogeneity, mobility, and energy-aware voronoi diagram in wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems 19*, 7 (jul. 2008), 995 –1008.

[3] Bahl, P., Chandra, R., Moscibroda, T., Murty, R., and Welsh, M. White space networking with wi-fi like connectivity. *ACM SIGCOMM* (2009).

[4] Chen, D., Yin, S., Zhang, Q., Liu, M., and Li, S. Mining spectrum usage data: a large-scale spectrum measurement study. In *ACM MobiCom* (2009).

[5] Chiang, R., Rowe, G., and Sowerby, K. A quantitative analysis of spectral occupancy measurements for cognitive radio. In *Vehicular Technology Conference* (2007).

[6] Ganesan, G., and Li, Y. Cooperative spectrum sensing in cognitive radio networks. In *DySpan* (2005), IEEE.

[7] Islam, M., Koh, C., Oh, S., Qing, X., Lai, Y., Wang, C., Liang, Y.-C., Toh, B., Chin, F., Tan, G., and Toh, W. Spectrum survey in singapore: Occupancy measurements and analyses. In *CrownCom* (2008).

[8] Iyer, A., Chintalapudi, K., Navda, V., Ramjee, R., Padmanabhan, V., and Murthy, C. Specnet: Spectrum sensing sans frontieres. Tech. rep., Microsoft Research, Feb 2011.

[9] McHenry, M. A. NSF Spectrum Occupancy Measurement Project Summary. In *Shared Spectrum Company Report* (2005).

[10] McHenry, M. A., Tenhula, P. A., McCloskey, D., Roberson, D. A., and Hood, C. S. Chicago spectrum occupancy measurements & analysis and a long-term studies proposal. In *TAPAS* (2006).

[11] Mishra, S. M., Sahai, A., and Brodersen, R. W. Cooperative Sensing Among Cognitive Radios. In *ICC* (2006), pp. 1658–1663.

[12] http://www.planet-lab.org/.

[13] PyInstaller. http://www.pyinstaller.org/.

[14] SCPI. http://www.ivifoundation.org/docs/SCPI-99.PDF.

[15] SpecNet Website. http://bit.ly/SpecNet.

[16] Unnikrishnan, J., and Veeravalli, V. Cooperative spectrum sensing and detection for cognitive radio. In *GLOBECOM* (2007), IEEE.

[17] Viera, M., Viera, L., Ruiz, L., Loureiro, A., Fernandes, A., and Nogueira, J. Scheduling nodes in wireless sensor networks: a voronoi approach. In *LCN* (2003), IEEE.

[18] Werner-Allen, G., Swieskowski, P., and Welsh, M. MoteLab: A Wireless Sensor Network Testbed. In *IPSN* (2005).

[19] Willcomm, D., Machiraju, S., Bolot, J., and Wolisz, A. Primary Users In Celular Networks : A Large Scale Measurement Study. In *DySPAN* (Oct 2008).

[20] Yang, L., Hou, W., Cao, L., Zhao, B. Y., and Zheng, H. Supporting demanding wireless applications with frequency-agile radios. In *NSDI* (2010), USENIX.

Towards Street-Level Client-Independent IP Geolocation

Yong Wang
UESTC/Northwestern University

Daniel Burgener
Northwestern University

Marcel Flores
Northwestern University

Aleksandar Kuzmanovic
Northwestern University

Cheng Huang
Microsoft Research

Abstract

A highly accurate client-independent geolocation service stands to be an important goal for the Internet. Despite an extensive research effort and significant advances in this area, this goal has not yet been met. Motivated by the fact that the best results to date are achieved by utilizing additional 'hints' beyond inherently inaccurate delay-based measurements, we propose a novel geolocation method that fundamentally escalates the use of external information. In particular, many entities (*e.g.*, businesses, universities, institutions) host their Web services locally and provide their actual geographical location on their Websites. We demonstrate that the information provided in this way, when combined with network measurements, represents a precious geolocation resource. Our methodology automatically extracts, verifies, utilizes, and opportunistically inflates such Web-based information to achieve high accuracy. Moreover, it overcomes many of the fundamental inaccuracies encountered in the use of absolute delay measurements. We demonstrate that our system can geolocate IP addresses 50 *times* more accurately than the best previous system, *i.e.*, it achieves a median error distance of 690 meters on the corresponding data set.

1 Introduction

Determining the geographic location of an Internet host is valuable for a number of Internet applications. For example, it simplifies network management in large-scale systems, helps network diagnoses, and enables location-based advertising services [17,24]. While coarse-grained geolocation, *e.g.*, at the state- or city-level, is sufficient in a number of contexts [19], the need for a *highly accurate* and reliable geolocation service has been identified as an important goal for the Internet (*e.g.,* [17]). Such a system would not only improve the performance of existing applications, but would enable the development of novel ones.

While client-assisted systems capable of providing highly accurate IP geolocation inferences do exist [3, 5, 9], many applications such as location-based access restrictions, context-aware security, and online advertising, can not rely on clients' support for geolocation. Hence, a highly accurate *client-independent* geolocation system stands to be an important goal for the Internet.

An example of an application that already extensively uses geolocation services, and would significantly benefit from a more accurate system, is online advertising. For example, knowing that a Web user is from New York is certainly useful, yet knowing the exact part of Manhattan where the user resides enables far more effective advertising, *e.g.*, of neighboring businesses. On the other side of the application spectrum, example services that would benefit from a highly accurate and dependable geolocation system, are the enforcement of location-based access restrictions and context-aware security [2]. Also of rising importance is cloud computing. In particular, in order to concurrently use public and private cloud implementations to increase scalability, availability, or energy efficiency (*e.g.*, [22]), a highly accurate geolocation system can help select a properly dispersed set of client-hosted nodes within a cloud.

Despite a decade of effort invested by the networking research community in this area, *e.g.*, [12, 15–19], and despite significant improvements achieved in recent years (*e.g.*, [17, 24]), the desired goal, a geolocation service that would actually enable the above applications, has not yet been met. On one hand, commercial databases currently provide rough and incomplete location information [17, 21]. On the other hand, the best result reported by the research community (to the best of our knowledge) was made by the Octant system [24]. This system was able to achieve a median estimation error of 22 miles (35 kilometers). While this is an admirable result, as we elaborate below, it is still insufficient for the above applications.

The key contribution of our paper lies in designing a novel client-independent geolocation methodology and

in deploying a system capable of achieving highly accurate results. In particular, we demonstrate that our system can geolocate IP addresses with a median error distance of 690 meters in an academic environment. Comparing to recent results on the same dataset shows that we improve the median accuracy by 50 times relative to [24] and by approximately 100 times relative to [17]. Improvements at the tail of the distribution are even more significant.

Our methodology is based on the following two insights. First, many entities host their Web services locally. Moreover, such Websites often provide the actual geographical location of the entity (*e.g.*, business and university) in the form of a postal address. We demonstrate that the information provided in this way represents a *precious* resource, *i.e.*, it provides access to a large number of highly accurate landmarks that we can exploit to achieve equally accurate geolocation results. We thus develop a methodology that effectively mines, verifies, and utilizes such information from the Web.

Second, while we utilize absolute network delay measurements to estimate the coarse-grained area where an IP is located, we argue that absolute network delay measurements are fundamentally limited in their ability to achieve fine-grained geolocation results. This is true in general even when additional information, *e.g.*, network topology [17] or negative constraints such as uninhabitable areas [24], is used. One of our key findings, however, is that *relative network delays* still heavily correlate with geographical distances. We thus fully abandon the use of absolute network delays in the final step of our approach, and show that a simple method that utilizes only relative network distances achieves the desired accuracy.

Combining these two insights into a single methodology, we design a three-tier system which begins at the large, coarse-grained scale, first tier where we utilize a distance constraint-based method to geolocate a target IP into an area. At the second tier, we effectively utilize a large number of Web-based landmarks to geolocate the target IP into a much smaller area. At the third tier, we opportunistically inflate the number of Web landmarks and demonstrate that a simple, yet powerful, closest node selection method brings remarkably accurate results.

We extensively evaluate our approach on three distinct datasets – Planetlab, residential, and an online maps dataset – which enables us to understand how our approach performs on an academic network, a residential network, and in the wild. We demonstrate that our algorithm functions well in all three environments, and that it is able to locate IP addresses in the real world with high accuracy. The median error distances for the three sets are 0.69 km, 2.25 km, and 2.11 km, respectively.

We demonstrate that factors that influence our system's accuracy are: (i) Landmark density, *i.e.*, the more landmarks there are in the vicinity of the target, the better accuracy we achieve. (ii) Population density, *i.e.*, the more people live in the vicinity of the target, the higher probability we obtain more landmarks, the better accuracy we achieve. (iii) Access technology, *i.e.*, our system has slightly reduced accuracy (by approximately 700 meters) for cable users relative to DSL users. While our methodology effectively resolves the last mile delay inflation problem, it is necessarily less resilient to the high last-mile latency *variance*, common for cable networks.

Given that our approach utilizes Web-based landmark discovery and network measurements on the fly, one might expect that the measurement overhead (crawling in particular) hinders its ability to operate in real time. We show that this is not the case. In a fully operational network measurement scenario, all the measurements could be done within 1-2 seconds. Indeed, Web-based landmarks are stable, reliable, and long lasting resources. Once discovered and recorded, they can be reused for many measurements and re-verified over longer time scales.

2 A Three-Tier Methodology

Our overall methodology consists of two major components. The first part is a three-tier active measurement methodology. The second part is a methodology for extracting and verifying accurate Web-based landmarks. The geolocation accuracy of the first part fundamentally depends on the second. For clarity of presentation, in this section we present the three-tier methodology by simply assuming the existence of Web-based landmarks. In the next section, we provide details about the extraction and verification of such landmarks.

We deploy the three-tier methodology using a distributed infrastructure. Motivated by the observation that the sparse placement of probing vantage points can avoid gathering redundant data [26], we collect 163 publicly available ping and 136 traceroute servers geographically dispersed at major cities and universities in the US.

2.1 Tier 1

Our final goal is to achieve a high level of geolocation precision. We achieve this goal gradually, in three steps, by incrementally increasing the precision in each step. The goal of the first step is to determine a coarse-grained region where the targeted IP is located. In an attempt not to 'reinvent the wheel,' we use a variant of a well established constrained-based geolocation (CBG) method [15], with minor modifications.

To geolocate the region of an IP address, we first send probes to the target from the ping servers, and convert the delay between each ping server and the target into a geographical distance. Prior work has shown that packets travel in fiber optic cables at 2/3 the speed of light in a vacuum (denoted by c) [20]. However, others have

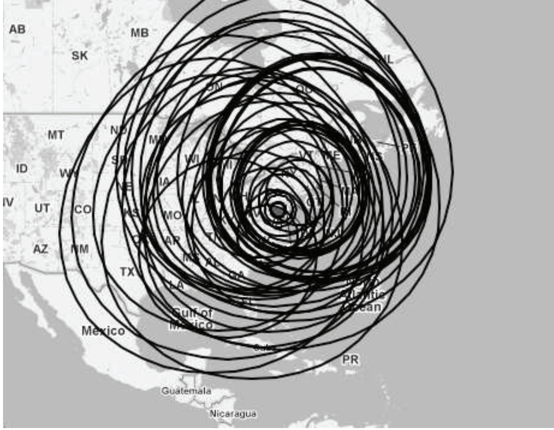


Figure 1: An example of intersection created by distance constraints

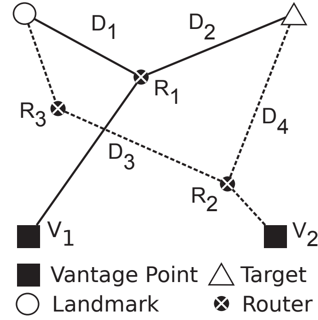


Figure 2: An example of measuring the delay between landmark and target

demonstrated that 2/3 c is a loose upper bound in practice due to transmission delay, queuing delay *etc*. [15, 17]. Based on this observation, we adopt 4/9 c from [17] as the converting factor between measured delay and geographical distance. We also demonstrate in Section 4, by using this converting factor, we are always capable of yielding a viable area covering the targeted IP.

Once we establish the distance from each vantage point, *i.e.*, ping server, to the target, we use multilateration to build an intersection that covers the target using known locations of these servers. In particular, for each vantage point, we draw a ring centered at the vantage point, with a radius of the measured distance between the vantage point and the target. As we show in Section 4, this approach indeed allows us to always find a region that covers the targeted IP.

Figure 1 illustrates an example. It geolocates a collected target (we will elaborate the way of collecting the targets in the wild in Section 4.1.2) whose IP address is 38.100.25.196 and whose postal address is '1850, K Street NW, Washington DC, DC, 20006'. We draw rings centered at the locations of our vantage points. The radius of each ring is determined by the measured distance between the vantage point (the center of this ring) and the target. Finally, we geolocate this IP in an area indicated by the shaded region, which covers the target, as shown in Figure 1.

Thus, by applying the CBG approach, we manage to geolocate a region where the targeted IP resides. According to [17, 24], CBG achieves a median error between 143 km and 228 km distance to the target. Since we strive for a much higher accuracy, this is only the starting point for our approach. To that end, we depart from pure delay measurements and turn to the use of external information available on the Web. Our next goal is to further determine a subset of ZIP Codes, *i.e.*, smaller regions that belong to the bigger region found via the CBG approach. Once we find the set of ZIP Codes, we will search for additional websites served within them. Our goal is to extract and verify the location information about these locally-hosted Web services. In this way, we obtain a number of accurate Web-based landmarks that we will use in Tiers 2 and 3 to achieve high geolocation accuracy.

To find a subset of ZIP Codes that belong to the given region, we proceed as follows. We first determine the center of the intersection area. Then, we draw a ring centered in the intersection center with a diameter of 5 km. Next, we sample 10 latitude and longitude pairs at the perimeter of this ring, by rotating by 36 degrees between each point. For the 10 initial points, we verify that they belong to the intersection area as follows. Denote by U the set of latitude and longitude pairs to be verified. Next, denote by V the set of all vantage points, *i.e.*, ping servers, with known location. Each vantage point v_i is associated with the measured distance between itself and the target, denoted by r_i. We wish to find all $u \in U$ that satisfy

$$\text{distance}(u, v_i) \leq r_i \text{ for all } v_i \in V$$

The distance function here is the great-circle distance [23], which takes into account the earth's sphericity and is the shortest distance between any two points on the surface of the earth measured along a path on the surface of the earth. We repeat this procedure by further obtaining 10 additional points by increasing the distance from the intersection center by 5 km in each round (*i.e.*, to 10 km in the second round, 15 km in the third *etc*.). The procedure stops when not a single point in a round belongs to the intersection. In this way, we obtain a sample of points from the intersection, which we convert to ZIP Codes using a publicly available service [4]. Thus, with the set of ZIP Codes belonging to the intersection, we proceed to Tier 2.

2.2 Tier 2

Here, we attempt to further reduce the possible region where the targeted IP is located. To that end, we aim to find Web-based landmarks that can help us achieve this goal. We explain the methodology for obtaining such landmarks in Section 3. Although these landmarks are passive, *i.e.*, we cannot actively send probes to other Internet hosts using them, we use the traceroute program to *indirectly* estimate the delay between landmarks and the target.

Learning from [11] that the more traceroute servers we use, the more direct a path between a landmark and the target we can find, we first send traceroute probes to the landmark (the empty circle in Figure 2) and the target (the triangle in Figure 2) from all traceroute servers (the solid squares V_1 and V_2 in Figure 2). For each vantage point, we then find the closest common router to the target and the landmark, shown as R_1 and R_2 in Figure 2, on the routes towards both the landmark and the target. Next, we calculate the latency between the common router and the landmark (D_1 and D_3 in Figure 2) and the latency between the common router and the target (D_2 and D_4 in Figure 2). We finally select the sum (D) of two latencies as the delay between landmark and target. In the example above, from V_1's point of view, the delay D between the target and the landmark is $D = D_1 + D_2$, while from V_2's perspective, the delay D is $D = D_3 + D_4$.

Since different traceroute servers have different routes to the destination, the common routers are not necessarily the same for all traceroute servers. Thus, each vantage point (a traceroute server) can estimate a different delay between a Web-based landmark and the target. In this situation, we choose the minimum delay from all traceroute servers' measurements as the final estimation of the latency between the landmark and the target. In Figure 2, since the path between landmark and target from V_1's perspective is more direct than that from V_2's ($D_1 + D_2 < D_3 + D_4$), we will consider the sum of D_1 and D_2 ($D_1 + D_2$) as the final estimation.

Routers in the Internet may postpone responses. Consequently, if the delay on the common router is inflated, we may underestimate the delay between landmark and target. To examine the 'quality' of the common router we use, we first traceroute different landmarks we collected previously and record the paths between any two landmarks, which also branch at that router. We then calculate the great circle distance [23] between two landmarks and compare it with their measured distance. If we observe that the measured distance is smaller than the calculated great circle distance for any pair of landmarks, we label this router as 'inflating', record this information, and do not consider its path (and the corresponding delay) for this or any other measurement.

Through this process, we can guarantee that the estimated delay between a landmark and the target is not underestimated. Nonetheless, such estimated delay, while converging towards the real latency between the two entities, is still usually larger. Hence, it can be considered as the upper bound of the actual latency. Using multilateration with the upper bound of the distance constraints, we further reduce the feasible region using the new tier 2 and the old tier 1 constraints.

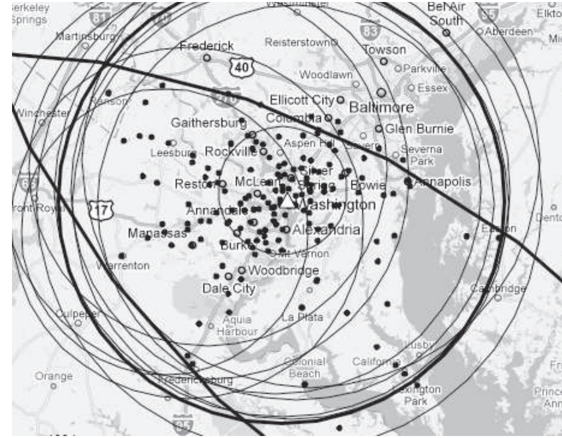


Figure 3: An example of shrinking the intersection

Figure 3 shows the zoomed-in subset of the constrained region together with old tier 1 constraints, marked by thick lines, and new tier 2 constraints, marked by thin lines. The figure shows a subset of sampled landmarks, marked by the solid dots, and the IP that we aim to geolocate, marked by a triangle. The tier 1 constrained area contains 257 distinctive ZIP Codes, in which we are able to locate and verify 930 Web-based landmarks. In the figure, we show only a subset of 161 landmarks for a clearer presentation. Some sampled landmarks lie outside the original tier 1 level intersection. This happens because the sampled ZIP Codes that we discover at the borders of the original intersection area typically spread outside the intersection as well. Finally, the figure shows that the tier 2 constrained area is approximately one order of magnitude smaller than the original tier 1 area.

2.3 Tier 3

In this final step, our goal is to complete our geolocation of the targeted IP address. We start from the region constrained in Tier 2, and aim to find all ZIP Codes in this region. To this end, we repeat the sampling procedure deployed in the Tier 2. This time from the center of the Tier 2 constrained intersection area, and at a higher granularity. In particular, we extend the radius distance by 1 km in each step, and apply a rotation angle of 10 degrees. Thus, we achieve 36 points in each round. We apply the same stopping criteria, *i.e.*, when no points in a round belong to the intersection. This finer-grain sam-

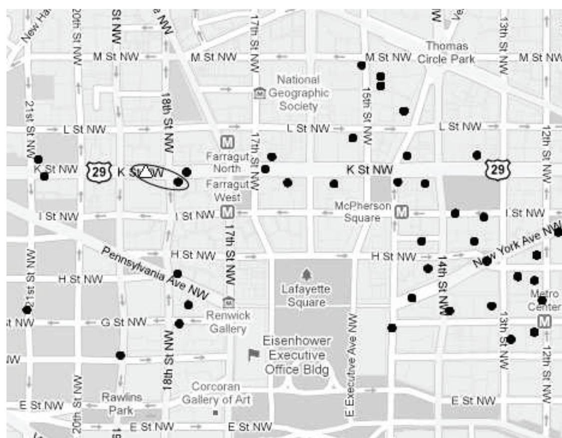


Figure 4: An example of associating a landmark with the target as the result

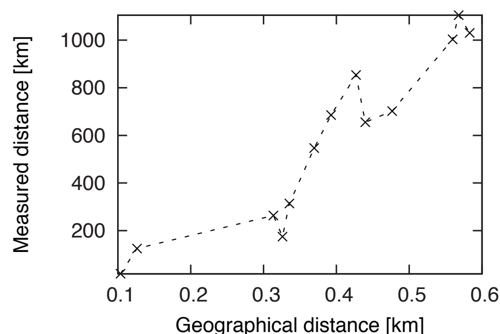


Figure 5: Measured distance vs. geographical distance.

pling process enables us to discover all ZIP Codes in the intersection area. For ZIP Codes that were not found in the previous step, we repeat the landmark discovery process (Section 3). Moreover, to obtain the distance estimations between newly discovered landmarks and the target, we apply the active probing traceroute process explained above.

Finally, knowing the locations of all Web-based landmarks and their estimated distances to the target, we select the landmark with the minimum distance to the target, and associate the target's location with it. While this approach may appear ad hoc, it signifies one of the key contributions of our paper. We find that on the smaller-scale, *relative distances* are preserved by delay measurements, overcoming many of fundamental inaccuracies encountered in the use of absolute measurements. For example, a delay of several milliseconds, commonly seen at the last mile, could place an estimate of a scheme that relies on absolute delay measurements hundreds of kilometers away from the target. On the contrary, selecting the closest node in an area densely populated with landmarks achieves remarkably accurate estimates, as we show below in our example case, and demonstrate systematically in Section 4 via large-scale analysis.

Figure 4 shows the striking accuracy of this approach. We manage to associate the targeted IP location with a landmark which is 'across the street', *i.e.*, only 0.103 km distant from the target. We analyze this result in more detail below. Here, we provide the general statistics for the Tier 3 geolocation process. In this last step, we discover 26 additional ZIP Codes and 203 additional landmarks in the smaller Tier 2 intersection area. We then associate the landmark, which is at '1776 K Street Northwest, Washington, DC' and has a measured distance of 10.6 km, yet a real geographical distance of 0.103 km, with the target. To clearly show the association, Figure 4 zooms into a very finer-grain street level in which the constrained rings and relatively more distant landmarks are not shown.

2.3.1 The Power of Relative Network Distance

Here, we explore how the relative network distance approach achieves such good results. Figure 5 sheds more light on this phenomenon. We examine the 13 landmarks within 0.6 km of the target shown in Figure 4. For each landmark, we plot the distance between the target and the Web-based landmarks (y-axis) (measured via the traceroute approach) as a function of the actual geographical distance between the landmarks and the target (x-axis). The first insight from the figure is that there is indeed a significant difference between measured distance, *i.e.*, their upper bounds, and the real distances. This is not a surprise. A path between a landmark, over the common router, to the destination (Figure 2) can often be circuitous and inflated by queuing and processing delays, as demonstrated in [17]. Hence, the estimated distance dramatically exceeds the real distance, by approximately three orders of magnitude in this case.

However, Figure 5 shows that the distance estimated via network measurements (y-axis) is largely *in proportion* with the actual geographical distance. Thus, despite the fact that the direct relationship between the real geographic distance and estimated distance is inevitably lost in inflated network delay measurements, the relative distance is largely preserved. This is because the network paths that are used to estimate the distance between landmarks and the target share vastly common links, hence experience similar transmission- and queuing-delay properties. Thus, selecting a landmark with the smallest delay is an effective approach, as we also demonstrate later in the text.

3 Extracting and Verifying Web-Based Landmarks

Many entities, *e.g.*, companies, academic institutions, and government offices, host their Web services locally. One implication of this setup is that the actual geographic addresses, (in the form of a street address, city, and ZIP Code), which are typically available at companies' and universities' home Web pages, correspond to the actual physical locations where these services are located. Accordingly, the geographical location of the corresponding web-servers' IP addresses becomes available, and the servers themselves become viable geolocation landmarks. Indeed, we have demonstrated above that such Web-based landmarks constitute an important geolocation resource. In this section, we provide a comprehensive methodology to automatically extract and verify such landmarks.

3.1 Extracting Landmarks

To automatically extract landmarks, we mine numerous publicly available mapping services. In this way, we are able to associate an entity's postal address with its domain name using such mapping services. Note that the use of online mapping services is a convenience, *not* a requirement for our approach. Indeed, the key resource that our approach relies upon is the existence of geographical addresses at locally hosted websites, which can be accessed directly at locally hosted websites.

In order to discover landmarks in a given ZIP Code, which is an important primitive of our methodology explained in Section 2 above, we proceed as follows. We first query the mapping service by a request that consists of the desired ZIP Code and a keyword, *i.e.*, 'business', 'university', and 'government office'. The service replies with a list of companies, academic institutions, or government offices within, or close to, this ZIP Code. Each landmark in the list includes the geographical location of this entity at the street-level precision and its web site's domain name.

As an example, a jewelry company at '55 West 47th Street, Manhattan, New York, NY, 10036', with the domain name www.zaktools.com, is a landmark for the ZIP Code 10036. For each entity, we also convert its domain name into an IP address to form a (domain name, IP address, and postal address) mapping. For the example above, the mapping in this case is (www.zaktools.com, 69.33.128.114, '55 West 47th Street, Manhattan, New York, NY, 10036'). A domain name can be mapped into several IP addresses. Initially, we map each of the IP addresses to the same domain name and postal address. Then, we verify all the extracted IP addresses using the methodology we present below.

3.2 Verifying Landmarks

A geographic address extracted from a Web page using the above approach may not correspond to the associated server's physical address for several reasons. Below, we explain such scenarios and propose verification methods to automatically detect and remove such landmarks.

3.2.1 Address Verification

The businesses and universities provided by online mapping services may be the landmarks *near* the areas covered by the ZIP Code, not necessarily *within* the ZIP Code. Thus, we first examine the ZIP Code in the postal address of each landmark. If a landmark has a ZIP Code different from the one we searched for, we remove it from the list of candidate landmarks. For example, for the ZIP Code 10036, a financial services company called Credit Suisse (www.credit-suisse.com) at '11 Madison Ave, New York, NY, 10010' is returned by online mapping services as an entity near the specified ZIP Code 10036. Using our verification procedure, we remove such a landmark from the list of landmarks associated with the 10036 ZIP Code.

3.2.2 Shared Hosting and CDN Verification

Additionally, a company may not always host its website locally. It may utilize either a CDN network to distribute its content or use shared hosting techniques to store its archives. In such situations, there is no one-to-one mapping between an IP address and a postal address in both CDN network and shared hosting cases. In particular, a CDN server may serve multiple companies' websites with distinct postal addresses. Likewise, in the shared hosting case a single IP address can be used by hundreds or thousands of domain names with diverse postal addresses. Therefore, for a landmark with such characteristics, we should certainly not associate its geographical location with its domain name, and in turn its IP address. On the contrary, if an IP address is solely used by a single entity, the postal address is much more trustworthy. While not necessarily comprehensive, we demonstrate that this method is quite effective, yet additional verifications are needed, as we explain in Section 3.2.3 below.

In order to eliminate a bad landmark, we access its website using (i) its domain name and (ii) its IP address independently. If the contents, or heads (distinguished by <head> and </head>), or titles (distinguished by <title> and </title>) returned by the two methods are the same, we confirm that this IP address belongs to a single entity. One complication is that if the first request does not hit the '$final$' content, but a redirection, we will extract the '$real$' URL and send an additional request to fetch the '$final$' content.

Take the landmark (www.manhattanmailboxes.com) at '676A 9 Avenue, New York, NY, 10036' as an ex-

ample. We end up with a web page showing ’access error’ when we access this website via its IP address, 216.39.57.104. Indeed, searching an online shared hosting check [8], we discover that there are more than 2,000 websites behind this IP address.

3.2.3 The Multi-Branch Verification

One final scenario occurs often in the real world: A company headquartered in a place where its server is also deployed may open a number of branches nationwide. Likewise, a medium size organization can also have its branch offices deployed locally in its vicinity. Each such branch office typically has a different location in a different ZIP Code. Still, all such entities have the same domain name and associated IP addresses as their headquarters.

As we explained in Section 2, we retrieve landmarks in a region covering a number of ZIP Codes. If we observe that some landmarks, with the same domain name, have different locations in different ZIP Codes, we remove them all. For example, the Allstate Insurance Company, with the domain name ’www.allstate.com’ has many affiliated branch offices nationwide. As a result, it shows up multiple times for different ZIP Codes in an intersection. Using the described method, we manage to eliminate all such occurrences.

3.3 Resilience to Errors

Applying the above methods, we can remove the vast majority of erroneous Web landmarks. However, exceptions certainly exist. One example is an entity (*e.g.*, a company) without any branch offices that hosts a website used exclusively by that company, but does not locate its Web server at the physical address available on the Website. In this case, binding the IP address with the given geographical location is incorrect, hence such landmarks may generate errors. Here, we evaluate the impact that such errors can have on our method’s accuracy. Counterintuitively, we show that the larger the error distance is between the claimed location (the street-level address on a website) and the real landmark location, the more resilient our method becomes to such errors. In all cases, we demonstrate that our method poses significant resilience to false landmark location information.

Figure 6 illustrates four possible cases for the relationship between a landmark’s real and claimed location. The figure denotes the landmark’s real location by an empty circle, the landmark’s claimed location by a solid circle, and the target by a triangle. Furthermore, denote $R1$ as the claimed distance, *i.e.*, the distance between the claimed location and the target. Finally, denote $R2$ as the measured distance between the landmark’s actual location and the target.

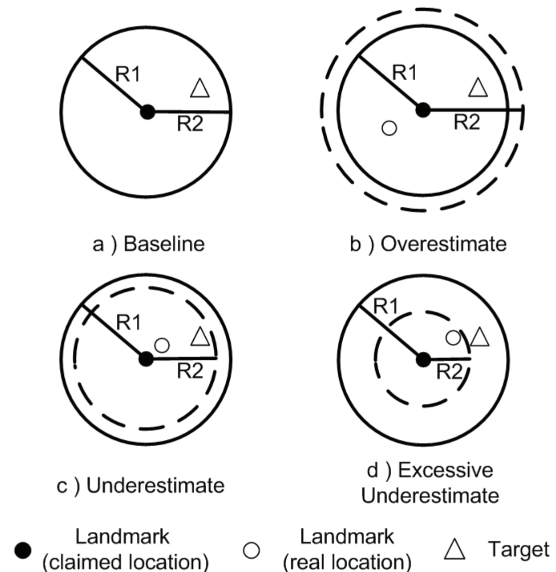


Figure 6: The effects of improper landmark

Figure 6(a) shows the baseline error-free scenario. In this case, the claimed and the real locations are identical. Hence, $R1 = R2$. Thus, we can draw a ring that is centered at the solid circle and is always able to contain the target, since the upper bound is used to measure the distance in Section 2.2.

Figure 6(b) shows the case when the claimed landmark’s location is different from the real location. Still, the real landmark is farther away from the target than the claimed location is. Hence, $R2 > R1$. Thus, we will draw a bigger ring with the radius of $R2$, shown as the dashed curve, than the normal case with the radius of $R1$. Thus, such an overestimate yields a larger coverage that always includes the target. Hence, our algorithm is unharmed, since the target remains in the feasible region.

Figures 6 (c) and (d) show the scenario when the real landmark’s location is closer to the target than the claimed location is, *i.e.*, $R2 < R1$. There are two sub scenarios here. In the underestimate case (shown in Figure 6(c)), the real landmark location is slightly closer to the target and the measured delay is only a little smaller than it should be. However, since the upper bound is used to measure the delay and convert it into distance, such underestimates can be counteracted. Therefore, we can still draw a ring with a radius of $R2$, indicated by the dashed curve, covering the target. In this case, the underestimate does not hurt the geolocation process.

Finally, in the excessive underestimate case (shown in Figure 6), the landmark is actually quite close to the target and the measured delay is much smaller than expected. Consequently, we end with a dashed curve with the radius of $R2$ that does not include the target, even when the upper bounds are considered. In this case, the

excessive underestimate leads us to an incorrect intersection or an improper association between the landmark and the target ($R2 < R1$). We provide a proof to demonstrate that the excessive underestimate case is not likely to happen in a technical report [10], yet we omit the proof here due to space constraints.

4 Evaluation

4.1 Datasets

We use three different datasets, Planetlab, residential, and online maps, as we explain below. Comparing with the large online maps dataset, the number of targets in the Planetlab and the residential datasets are relatively small. However, these two datasets help us gain valuable insights about the performance of our method in different environments, since the online maps dataset can contain both types of targets.

4.1.1 Planetlab dataset

One method commonly used to evaluate the accuracy of IP geolocation systems is to geolocate Planetlab nodes, *e.g.*, [17, 24]. Since the locations of these nodes are known publicly (universities must report the locations of their nodes), it is straightforward to compare the location given by our system with the location provided by the Planetlab database. We select 88 nodes from Planetlab, limiting ourselves to at most one node per location. Others (*e.g.*, [17]) have observed errors in the given Planetlab locations. Thus, we manually verify all of the nodes locations.

4.1.2 Residential dataset

Since the set of Planetlab nodes are all located on academic networks, we needed to validate our approach on residential networks as well. Indeed, many primary applications of IP geolocation target users on residential networks. In order to do this, we created a website, which we made available to our social networks, widely dispersed all over the US. The site automatically records users' IP addresses and enables them to enter their postal address and the access provider. In particular, we enable six selections for the provider: AT&T, Comcast, Verizon, other ISPs, University, and Unknown. Moreover, we explicitly request that users not enter their postal address if they are accessing this website via proxy, VPN, or if they are unsure about their connection. We then distribute the link to many people via our social networks, and obtained 231 IP address and location pairs.

Next, we eliminate duplicate IPs, 'dead' IPs that are not accessible over the course of the experiment, which is one-month after the data was collected. We also eliminate a large number of IPs with access method 'university' or 'unknown', since we intend to extract residential IPs and compare with those of academic IPs in Section 4.2. After elimination, we are left with 72 IPs.

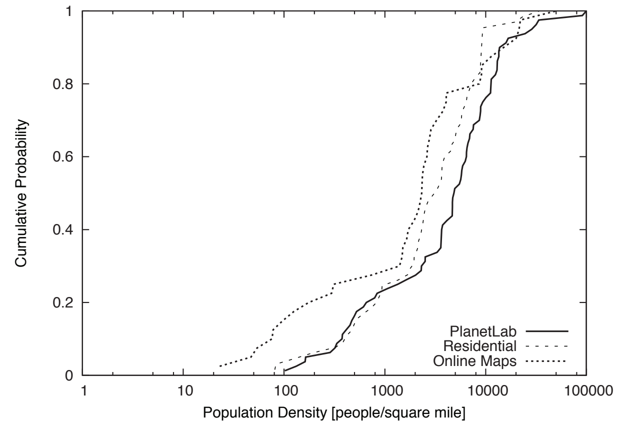


Figure 7: The distribution of the population density of three datasets

4.1.3 Online Maps dataset

We obtained a large-scale query trace from a popular online maps service. This dataset contains three-months of users' search logs for driving directions.[1] Each record consists of the user access IP address, local access time at user side, user browser agent, and the driving sequence represented by two pairs of latitude and longitude points. Our hypothesis here is that if we observe a location, as either source or destination in the driving sequence, periodically associated with an IP address, then this IP address is likely at that location. To extract such association from the dataset, we employ a series of strict heuristics as follows.

We first exclude IP addresses associated with multiple browser agents. This is because it is unclear whether this IP address is used by only one user with multiple browsers or by different users. We then select IP addresses for which a single location appears at least four times in each of the three months, since such IP addresses with 'stable' search records are more likely to provide accurate geolocation information than the ones with only a few search records. We further remove IP addresses that are associated with two or more locations that appear at least four times. Finally we remove all 'dead' IPs from the remaining dataset.

4.1.4 Dataset characteristics

Here, our goal is to explore the characteristics of the locations where the IP addresses of the three datasets are.

[1] We respect a request of this online map service company and do not disclose the number of requests and collected IPs here and in the rest of the paper.

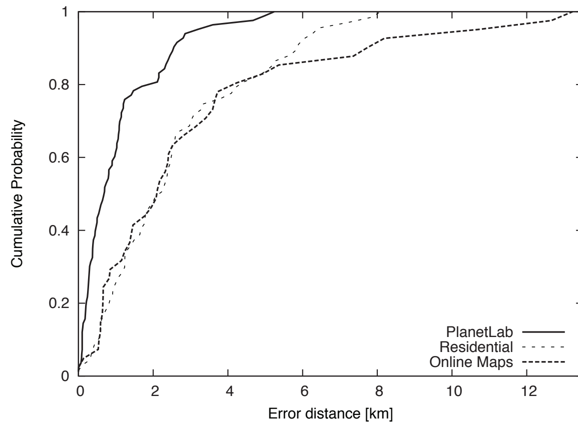


Figure 8: Comparison of error distances of three datasets

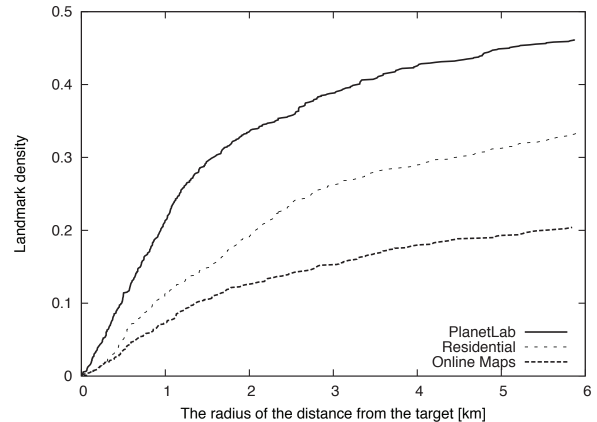


Figure 9: Landmark density of three datasets

In particular, population density is an important parameter that indicates the rural vs. urban nature of the area in which an IP address resides. We will demonstrate below that this parameter influences the performance of our method, since urban areas typically have a large number of web-based landmarks.

Figure 7 shows the distribution of the population density of the ZIP Code at which the IP addresses of the three datasets locate. We obtain the population density for each ZIP Code by querying the website City Data [1]. Figure 7 shows that our three datasets cover both rural areas, where the population density is small, and urban areas, where the population density is large. In particular, all three datasets have more than 20% of IPs in ZIP Codes whose population density is less than 1,000. The figure also shows that PlanetLab dataset is the most 'urban' one, while the Online Maps datasets has the longest presence in rural areas. In particular, about 18% of IPs in the Online Maps dataset reside in ZIP Codes whose population density is less than 100.

4.2 Experimental results

4.2.1 Baseline results

Figure 8 shows the results for the three datasets. In particular, it depicts the cumulative probability of the error distance, *i.e.*, the distance between a target's real location and the one geolocated by our system. Thus, the closer the curve is to the upper left corner, the smaller the error distance, and the better the results. The median error for the three datasets, a measure typically used to represent the accuracy of geolocation systems [15, 17, 24], are 0.69 km for Planetlab, 2.25 km for the residential dataset, and 2.11 km for the online maps dataset. Beyond excellent median results, the figure shows that the tail of the distribution is not particularly long. Indeed, the maximum error distances are 5.24 km, 8.1 km, and 13.2 km for Planetlab, residential, and online maps datasets, respectively. The figure shows that the performances of the residential and online maps datasets are very similar. This is not a surprise because the online maps dataset is dominated by residential IPs. On the other hand, our system achieves clearly better results in the Planetlab scenario. We analyze this phenomenon below.

4.2.2 Landmark density

Here, we explore the number of landmarks in the proximity of targeted IPs. The larger the number of landmarks we can discover in the vicinity of a target, the larger the probability we will be able to more accurately geolocate the targeted IP. We proceed as follows. First, we count the number of landmarks in circles of radius r, which we increase from 0 to 6 km, shown in Figure 9. Then, we normalize the number of landmarks for each radius relative to the total number of landmarks seen by all three datasets that fit into the 6 km radius. Because of such normalization, the normalized number of targets for $x = 6km$ sum up to 1. Likewise, due to normalization, the value on y-axis could be considered the landmark density.

Figure 9 shows the landmark density for the three datasets as a function of the radius. The figure shows that the landmark density is largest in the Planetlab case. This is expected because one can find a number of Web-based landmarks on a University campus. This certainly increases the probability of accurately geolocating IPs in such an environment, as we demonstrated above. The figure shows that residential targets experience a lower landmark density relative to the Planetlab dataset. At the same time, the online maps dataset shows an even lower landmark density. As shown in Figure 7, our residential dataset is more biased towards urban areas. On the contrary, the online maps provide a more comprehensive and unbiased breakdown of locations. Some of them are

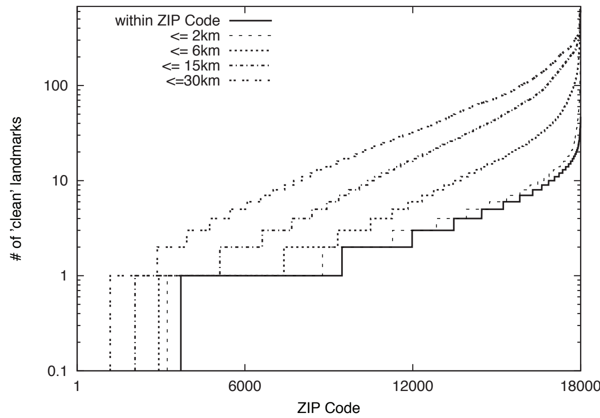


Figure 10: Number of clean landmarks for each ZIP Code

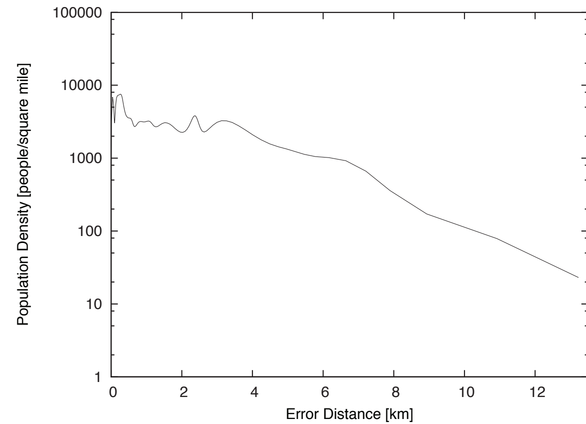


Figure 11: Error distance vs. population density

rural areas, where the density of landmarks is naturally lower. In summary, the landmark density is certainly a factor that clearly impacts our system's geolocation accuracy. Still, additional factors such as access network level properties do play a role, as we show below.

4.2.3 Global landmark density

To understand the global landmark density (more precisely, US-wide landmark density), we evenly sample 18,000 ZIP Codes over all states in US. Figure 10 shows that there are 79.4% ZIP Codes which contain at least one landmark within the ZIP Code. We manually check the remaining ZIP Codes and realize that they are typically the rural areas, where local entities, *e.g.,* businesses, are rare naturally. Nonetheless, for 83.78% of ZIP Codes, we are capable of finding out at least one landmark in its vicinity of 6 km; for 88.51% of ZIP Codes, we are always able to discover at least one landmark in its vicinity of 15 km; finally, for 93.44% of ZIP Codes, we find at least one landmark in its vicinity of 30 km.

We make the following comments. First, Figure 10 can be used to predict US-wide performance of our method from the *area* perspective. For example, it shows that for 6.6% of the territory, the error can only be larger than 30 km. Note, however, that such areas are extremely sparsely populated. For example, the average population density in the 6.6% of ZIP Codes that have no landmark within 30 km is less than 100. Extrapolating conservatively to the entire country, it can be computed that such areas account for about 0.92% of the entire population.

4.2.4 The role of population density

Here, we return to our datasets and evaluate our system' s performance, *i.e.*, error distance, as a function of population density. For the sake of clarity, we merge the results of the three datasets. Figure 11 plots the best fit curve that captures the trends. It shows that the error distance is smallest in densely populated areas, while the error grows as the population density decreases. This result is in line with our analysis in Section 4.2.3. Indeed, the larger the population density is, the higher probability we can discover more landmarks. Likewise, as shown in Section 4.2.2, the more landmarks we can discover in the vicinity of targeted IP address, the higher probability we can more accurately geolocate the targeted IP. Finally, the results show that our system is still capable of geolocating IP addresses in rural areas as well. For example, we trace the IP that shows the worst error of 13.2 km. We find that this is an IP in a rural area with no landmarks discovered within the ZIP Code, which has a population density of 47. The landmark with the minimum measured distance is 13.2 km away, which our system selected.

4.2.5 The role of access networks

Contrary to the academic environment, a number of residential IP addresses access the Internet via DSL or cable networks. Such networks create the well-known last-mile delay inflation problem, which represents a fundamental barrier to methods that rely on absolute delay measurements. Because our method relies on *relative* delay measurements, it is highly resilient to such problems, as we show below. To evaluate this issue, we examine and compare our system's performance for three different residential network providers that we collected in Section 4.1.2. These are AT&T, Comcast, and Verizon.

Figure 12 shows the CDF of the error distance for the three providers. The median error distance is 1.48 km for Verizon, 1.68 km for AT&T, and 2.38 km for Comcast. Thus, despite the fact that we measure significantly inflated delays in the last mile, we still manage to geolo-

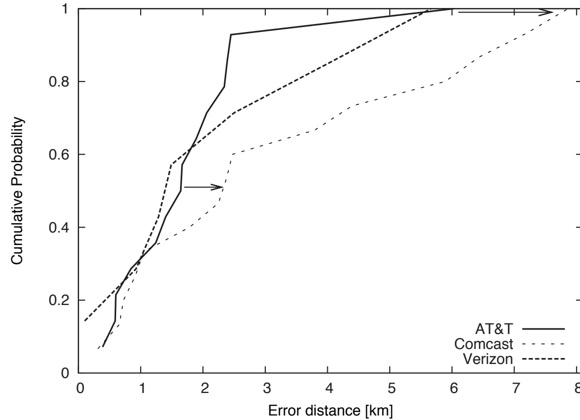


Figure 12: Comparisons of error distance in different ISPs

cate the endpoints very accurately. For example, a delay of 5 ms [13] that we commonly see at the last mile could place a scheme relying on absolute delay measurements 700 km away from the target. Our approach effectively addresses this problem and geolocates the targets within a few kilometers.

Figure 12 shows that our method has reduced performance for Comcast targets, who show a somewhat longer tail than the other two providers. We explore this issue in more depth. According to [7], AT&T and Verizon offer DSL services. Comcast is dominantly a cable Internet provider, and offers DSL only in a smaller number of areas. As demonstrated in [13], cable access networks have a much larger latency *variance*, which may rapidly vary over short time scales, than DSL networks. While our relative delay approach is resilient to *absolute* delay inflation at the last mile, it can still be hurt by measured delay variance. Because latency in cable networks changes over short time scales, it blurs our measurements, which are not fully synchronized. Hence, the landmarks' relative proximity estimation gets blurred, which causes the effects shown in the figure. In particular, the median error distance of the cable case increases by approximately 700 meters relative to the DSL case (shown by the arrow from AT&T to Comcast in the middle of Figure 12), while the maximum error distance increases by 2 km (shown by the arrow from AT&T to Comcast at the top of Figure 12).

5 Discussion

Measurement overhead. Our methodology incurs measurement overhead due to Web crawling and network probing. Still, it is capable of generating near real-time responses, as we explain below. To geolocate an IP address, we crawl Web landmarks for a portion of ZIP Codes on the fly, as we explained in Sections 2.2 and 2.3. It is important to understand that this is a *one-time* overhead per ZIP Code because we cache all landmarks for every ZIP Code that we visit. Thus, when we want to geolocate other IP addresses in the vicinity of a previous one, we reuse previously cached landmarks. Once this dataset is built, only occasional updates are needed. This is because the Web-based landmarks we use are highly stable and long-lived in the common case.

On the network measurement side, we generate concurrent probes from multiple vantage points simultaneously. In the first tier, we need 2 RTTs (1 RTT from the master node to the vantage points, and 1 RTT for the ping measurements). In the second and third tiers each, the geolocation response time per IP can be theoretically limited by 3 round-trip times (1 RTT from the master node to the measurement vantage points, and 2 RTTs for an advanced traceroute overhead[2]). Thus, the total overhead on the network measurement side is 8 RTTs, which typically translates to a 1-2 seconds delay.

Migrating web services to the cloud. Cloud services are thriving in the Internet. One might have a concern that this might dramatically reduce the number of landmarks that we can rely upon. We argue that this is not the case. While more websites might indeed be served on the cloud, the total number of websites will certainly increase over time. Even if the large percent of the websites will end up in the cloud, the remaining percent of websites will always create a reliable and accurate backbone for our method. Moreover, even when an entity migrates a Web site to the cloud, the associated e-mail exchange servers do remain hosted locally (results not shown here due to space constraints). Hence, such servers can serve as accurate geolocation landmarks. Our key contribution lies in demonstrating that all such landmarks (*i.e.*, Web, e-mail, or any other) can be effectively used for accurate geolocation.

International coverage. Our evaluation is limited to US simply as we were able to obtain the vast majority of the ground-truth information from within the US. Still, we argue that our approach can be equally used in other regions as well. This is because other countries such as Canada, UK, China, India, South Korea *etc.*, also have their own "ZIP Code" systems. We are currently adjusting our system so that it can effectively work in these countries. Moreover, we expect that our approach will be applicable even in regions with potentially poor network connectivity. This is because our relative-delay-based method is insensitive to inflated network latencies characteristic for such environments.

[2] In the advanced traceroute case, 1 RTT is needed to obtain the IPs of intermediate routers, while another RTT is needed to simultaneously obtain round-trip time estimates to all intermediate routers by sending concurrent probes.

6 Related work

6.1 Client-independent IP geolocation systems

6.1.1 Data mining-based

DNS-based. Davis *et al*. [12] propose a DNS-based approach, which suggests adding location segments in the format of a Resource Record (RR). Nevertheless, such modification can not be easily deployed in practice and the administrators have little incentive to register or modify new RRs. Moreover, Zhang *et al*. [25] have demonstrated that DNS misnaming is common, and that it can distort Internet topology mapping.

Whois-based. Moore *et al*. [18] argue that geolocation can also be obtained by mining the Whois database. However, as the authors themselves pointed out, large entities with machines dispersed in different locations can register their domain names with the geographical location of their headquarters. As an example, many existing IP geolocation databases that use this approach incorrectly locate all Google's servers worldwide to Mountain View, CA.

Hostname-based. The machine hostnames can sometimes indicate the geolocation information. In particular, Padmanabhan's and Subramanian's GeoTrack [19] parses the location of the last access router towards the target to be located from its hostname and uses the location of this router as that of the target. Unfortunately, this method can be inhibited by several factors, as pointed by [14]. First, not all machine names contain geolocation associating information. Second, administrators can be very creative in naming the machines; hence, parsing all kinds of formats becomes technically difficult. Finally, such last hop location substitution can incur errors.

Web-based. Guo *et al*.'s [16]'s Structon, mines the geolocation information from the Web. In particular, Structon builds a geolocation table and uses regular expressions to extract location information from each web page of a very large-scale crawling dataset. Since Structon does not combine delay measurement with the landmarks it discovers, it achieves a much coarser (city-level) geolocation granularity. For example, they extract all location keywords from a web page rather than just the location address. Likewise, they geolocate a domain name by choosing one from all locations provided by all the web pages within this domain name. Indeed, such approaches are error prone. Moreover, geolocating a /24 segment with a city blurs the finer-grained characteristics of each IP address in this segment.

Other sources. Padmanabhan's and Subramanian's GeoCluster [19] geolocates IP addresses into a geographical cluster by using the address prefixes in BGP routing tables. In addition, by acquiring the geolocation information of some IP addresses in a cluster from proprietary sources, *e.g*., users' registration records in the Hotmail service, GeoCluster deduces the location of this entire cluster. This method highly depends on the correctness of users' input and the private location information, which is in general not publicly available. Our approach differs from GeoCluster in that web designers have strong incentive to report correct location information in their websites, while users are less likely to provide accurate location information in their registration application with online services, on which GeoCluster highly relies. Moreover, we have demonstrated that using active network measurements instead of extrapolating geo information to entire clusters, is far more accurate.

6.1.2 Delay measurement-based

GeoPing. Padmanabhan and Subramanian design GeoPing [19], which assumes that two machines that have similar delay vectors tend to be close to each other. The authors rely on a set of active landmarks, *i.e*., those capable of actively probing the target. Necessarily, the accuracy of such an approach (the comparable results shown later in the text) depends on the number of active landmarks, which is typically moderate.

CBG. Instead of yielding a discrete single geo point, Gueye *et al*. [15] introduce Constraint Based Geolocation (CBG), a method that provides a continuous geo *space* by using multilateration with distance constraints. In particular, CBG first measures the delays from all vantage points to the target. Then, it translates delays into distance by considering the best network condition of each vantage point, termed *bestline*. Finally, it returns a continuous geo space by applying multilateration.

CBG uses bestline constraints to compensate for the fact that Internet routes are sometimes undirected or inflated. However, due to the difficulty of predicting the directness of a network route from a vantage point to a target, CBG only works well when the target is close to one of the vantage points. As explained above, we use the CBG approach straightforwardly in our tier 1 phase to discover the coarse-grained area for a targeted IP. Moreover, using newly discovered web landmarks in this area, we further constrain the targeted area in the tier 2 phase as well. Thus, while CBG is good at limiting the destination area, it is inherently limited in its ability to achieve very fine-grained resolution due to measurement inaccuracies.

TBG. Taking the advantage of the fact that routers close to the targets can be more accurately located, Katz-Bassett *et al*. [17] propose Topology-based Geolocation (TBG), which geolocates the target as well as the routers in the path towards the target. The key contribution of this work lies in showing that network topology can be effectively used to achieve higher geolocation accuracy. In particular, TBG uses the locations of routers in the in-

terim as landmarks to better quantify the directness of the path to the target and geolocate it.

In addition to using network topological information, a TBG variant also takes advantage of passive landmarks with known locations. However, such an approach is constrained by the fact that it only has a very limited number of such landmarks. On the contrary, our web-based technique can conquer this difficulty significantly by discovering a large number of web-based landmarks. More substantially, TBG fundamentally relies on the *absolute* delay measurements, which are necessarily inaccurate at short distances. On the contrary, in addition to relying on a large number of web-based landmarks in an area, we demonstrate that our relative distance approach, while technically less attractive, is far more accurate.

Octant. Wong *et al*. [24] propose Octant, which considers the locations of intermediate routers as landmarks to geolocate the target. Further, Octant considers both positive information, the maximum distance that a target may be from the landmark, and negative information, the minimum distance this target may be from the landmark. In addition to delay-based constraints, Octant also enables any kind of positive and negative constraints to be deployed into its system, *e.g*., the negative constraints (oceans and uninhabitable areas) obtained from geography and demographics.

In attempt to achieve high accuracy, Octant (as well as the above TBG method) also adopts the locations of routers in the path to the destination as landmarks to geolocate the target. However, such an approach is hampered to reach finer-grained accuracy because it fails to accurately geolocate routers at such precision in the first place. Finally, while Octant 'pushes' the accuracy of delay-based approaches to an absolutely admirable limit, it is incapable of achieving a higher precision simply due to the inherent inaccuracies associated with absolute delay measurements.

Comparative results. According to [17], TBG has the median estimation error of 67 km that a factor of three outperforms CBG with the median estimation error of 228 km. According to [24], comparing with GeoPing and CBG, Octant with a median estimation error of 22 miles is three times better than GeoPing with an estimation error of 68 miles and four times better than CBG with an error distance of 89 miles respectively. Because TBG and Octant used the PlanetLab nodes to evaluate their system's accuracy, we can directly compare them with our system. As outlined above, our system's median error distance is 50 times smaller than Octant's, and approximately 100 times smaller than TBG's.

6.2 Client-dependent IP geolocation systems

6.2.1 Wireless geolocation

GPS-based geolocation Global Positioning System (GPS) devices, that have been embedded into billions of mobile phones and computers at nowadays, could precisely provide user's location. However, GPS technology differs from our geolocation strategy in the sense that it is a 'client-side' geolocation approach, which means that the server does not know where the user is, unless the user explicitly reports his information back to the server.

Cell tower and Wi-Fi -based geolocation. Google My Location [5] and Skyhook [9] introduced their cell tower-based and Wi-Fi -based geolocation approaches. In particular, the cell tower-based geolocation offers users estimated locations by triangulating from cell towers surrounding users, while the Wi-Fi-based geolocation uses Wi-Fi access point information instead of cell towers. Specifically, every tower or Wi-Fi access point has a unique identification and footprint. To find a user's approximate location, such methods calculate user's position relative to the unique identifications and footprints of nearby cell towers or Wi-Fi access points.

Such methods could provide accurate results, *e.g*., 200 - 1000 meters accuracy in cell tower scenario, and 10-20 meters in Wi-Fi scenario [9], on the expense of sacrificing the geolocation availability at three aspects.

First, these approaches require end user's permission to share their location. However, as we discussed above, many applications such as location-based access restrictions, context-aware security, and online advertising, can not rely on client's support for geolocation. Second, companies utilizing such an approach must deploy drivers to survey every single street and alley in tens of thousands of cities and towns worldwide, scanning for cell towers and Wi-Fi access points, as well as plotting their geographic locations. However, in our approach, we avoid such 'heavy' overhead by lightly crawling landmarks from the Web. Third, these approaches are tailored towards mobile phones and laptops. However, there are many devices (IPs) bound with wired network on the Internet. Such wireless geolocation methods are necessarily incapable of geolocating these IPs, while our method does not require any precondition on the end devices and IPs.

6.2.2 W3C geolocation

A geolocation API specification [3] is going to become a part of HTML 5 and appears to be a part of current browsers already [6].This API defines a high-level interface to location information, and is agnostic of the underlying location information sources. The underlying location database could be collected and calculated

by GPS, Wi-Fi access point, cell tower, RFID, Bluetooth MAC address, as well as IP address, associated with the devices. Again, this approach requires end users' collaboration for geolocation. In addition, this method also requires browser compatibility, *e.g.*, Web browser must supports HTML 5. Finally, to geolocate wired devices, W3C geolocation has to conduct IP address-based approaches discussed in Section 6.1.1 and Section 6.1.2. In this case, our method can be considered as an effective alternative to improve the accuracy.

7 Conclusions

We have developed a client-independent geolocation system able to geolocate IP addresses with more than an order of magnitude better precision than the best previous method. Our methodology consisted of two powerful components. First, we utilized a system that effectively harvest geolocation information available on the Web to build a database of landmarks in a given ZIP Code. Second, we employed a three tiered system that begins at a large, coarse-grained, scale and progressively works its way to a finer, street-level, scale. At each stage, it takes advantage of landmark data and the fact that on the smaller-scale, relative distances are preserved by delay measurements, overcoming many of fundamental inaccuracies encountered in the use of absolute measurements. By combining these we demonstrated the effectiveness of using both active delay measurements and web-mining for geo-location purposes.

We have shown that our algorithm functions well in the wild, and is able to locate IP addresses in the real world with extreme accuracy. Additionally, we demonstrated that our algorithm is widely applicable to IP addresses from both academic institutions, a collection of residential addresses, as well as a larger mixed collection of addresses. The high accuracy of our system in a wide range of networking environments demonstrates its potential to dramatically improve the performance of existing location-dependent Internet applications and to open the doors to novel ones.

References

[1] City data. http://www.city-data.com/.

[2] Geolocation and application delivery. www.f5.com/pdf/white-papers/geolocation-wp.pdf.

[3] Geolocation api specification. http://dev.w3.org/geo/api/spec-source.html.

[4] Geonames. http://www.geonames.org/.

[5] Google maps with my location. http://www.google.com/mobile/gmm/mylocation/index.html.

[6] How google maps uses the w3c geolocation api and google location services. http://apb.directionsmag.com/archives/6094-How-Google-Maps-uses-the-W3C-Geolocation/-API-and-Google-Location-Services.html.

[7] Ookla's speedtest throughput measures. http://silicondetector.org/display/IEPM/Ookla%27s+Speedtest+Throughput+Measures.

[8] Reverse ip domain check. http://www.yougetsignal.com/tools/web-sites-on-web-server/.

[9] Skyhook. http://www.skyhookwireless.com/.

[10] Technical report. http://networks.cs.northwestern.edu/technicalreport.pdf.

[11] CHENG, H., ANGELA, W., JIN, L., AND W, R. K. Measuring and evaluating large-scale CDNs. In *Microsoft Technical Report*.

[12] DAVIS, C., VIXIE, P., GOODWIN, T., AND DICKINSON, I. A means for expressing location information in the domain name system. *RFC 1876* (1996).

[13] DISCHINGER, M., HAEBERLEN, A., GUMMADI, K. P., AND SAROIU, S. Characterizing residential broadband networks. In *IMC, '07*.

[14] FREEDMAN, M. J., VUTUKURU, M., FEAMSTER, N., AND BALAKRISHNAN, H. Geographic locality of ip prefixes. In *IMC, '05*.

[15] GUEYE, B., ZIVIANI, A., CROVELLA, M., AND FDIDA, S. Constraint-based geolocation of internet hosts. *Transactions on Networking* (2006).

[16] GUO, C., LIU, Y., SHEN, W., WANG, H. J., YU, Q., AND ZHANG, Y. Mining the web and the internet for accurate ip address geolocations. In *Infocom mini conference, '09*.

[17] KATZBASSETT, E., JOHN, J. P., KRISHNAMURTHY, A., WETHERALL, D., ANDERSON, T., AND YATIN. Towards ip geolocation using delay and topology measurements. In *IMC, '06*.

[18] MOORE, D., PERIAKARUPPAN, R., DONOHOE, J., AND CLAFFY, K. Where in the world is netgeo.caida.org? In *INET '00*.

[19] PADMANABHAN, V. N., AND SUBRAMANIAN, L. An investigation of geographic mapping techniques for internet host. In *ACM SIGCOMM '01*.

[20] PERCACCI, R., AND VESPIGNANI, A. Scale-free behavior of the internet global performance. *The European Physical Journal B - Condensed Matter* (2003).

[21] SIWPERSAD, S., BAMBAGUEYE, AND UHLIG, S. Assessing the geographic resolution of exhaustive tabulation for geolocating internet hosts. In *PAM, '08*.

[22] VALANCIUS, V., LAOUTARIS, N., MASSOULIE, L., DIOT, C., AND RODRIGUEZ, P. Greening the Internet with nano data centers. In *CONEXT '09*.

[23] VINCENTY, T. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey Review* (1975).

[24] WONG, B., STOYANOV, I., AND SIRER, E. G. Octant: A comprehensive framework for the geolocalization of internet hosts. In *NSDI, '07*.

[25] ZHANG, M., RUAN, Y., PAI, V., AND REXFORD, J. How dns misnaming distorts internet topology mapping. In *USENIX Annual Technical Conference, '06*.

[26] ZIVIANI, A., FDIDA, S., DE REZENDE, J. F., AND DUARTE, O. C. M. Improving the accuracy of measurement-based geographic location of internet hosts. *Computer Networks, Elsevier Science* (2005).